Tema 14

Imágenes en Java

Una imagen es un arreglo bidimensional de colores. Cada elemento de un arreglo se llama pixel. Una imagen es diferente de una superficie de dibujo. Los pixeles de una imagen no coinciden necesariamente con un pixel en la superficie de dibujo. Una imagen tiene ancho y un alto, medido en pixeles y un sistema de coordenadas que es independiente de la superficie de dibujo.

En Java 2D, una imagen se representa por las clases `Image` y `BufferedImage`, figura 14.1. La clase abstracta `Image` representa una imagen como un arreglo rectangular de pixeles. La clase `BufferedImage` le permite a la aplicación trabajar directamente con los datos de la imagen (por ejemplo obteniendo o estableciendo el color de un pixel). Esta clase administra directamente la imagen en memoria y provee métodos para almacenar, interpretar y obtener datos de los pixeles.

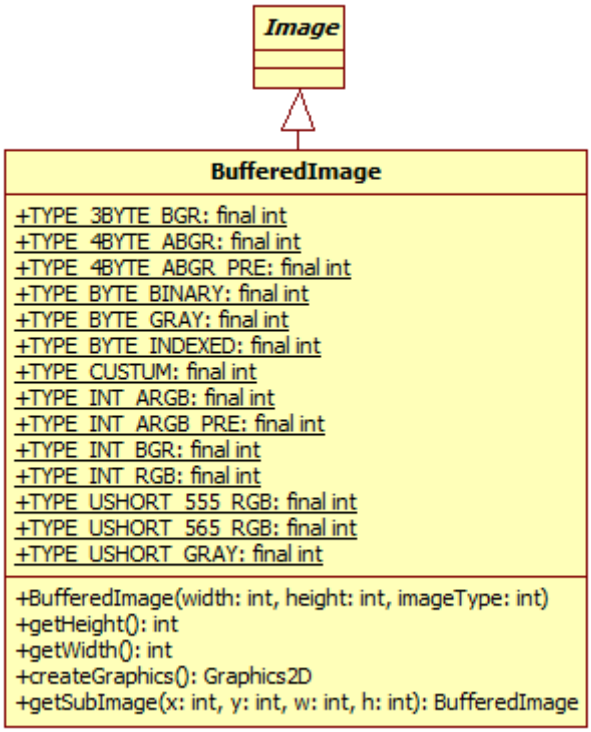

Figura 14.1 Clases que Representan una Imagen

Un objeto de tipo `BufferedImage` es esencialmente un objeto del tipo `Image` con un buffer de datos accesible, por lo que la hace más eficiente para trabajar que con un objeto del tipo `Image`. Un objeto de tipo `BufferedImage` tiene un modelo de color, un objeto del tipo `ColorModel` y un entramado de los datos de la imagen, un objeto del tipo `Raster`, figura 14.2.

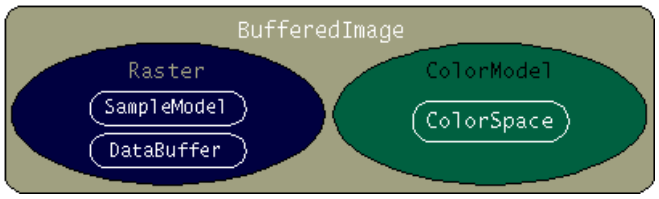

Figura 14.2 Un objeto de Tipo Buffered Image

El objeto de tipo `ColorModel` provee una interpretación de los pixeles de la imagen dentro del espacio de color. Un espacio de color es esencialmente una colección de todos los colores que pueden representarse en un dispositivo particular. Los monitores, por ejemplo, por lo general definen su espacio de color usando el color de espacio GB. Una impresora, por otro lado puede usar el espacio de color CMYK. Las imágenes pueden usar una de varias subclases de `ColorModel` en las bibliotecas de la API de java 2D:

- `ComponentColorModel`, en la que un pixel se representa por varios valores discretos, tipicamente bytes, cada uno representando una componente del color, como el componente rojo en la representación RGB.
- `DirectColorModel`, en la que todos los componentes de un color están empaquetados juntos en bits separados del mismo valor de un pixel.
- `IndexColorModel`, en la que cada pixel es un valor representado como un índice en una paleta de colores.

El objeto de tipo `Raster` almacena los datos reales de los pixeles para una imagen en un arreglo rectangular accesado por las coordenadas x, y. Está formado de dos partes:

- Un buffer de datos, que contiene los datos crudos de la imagen.
- Un modelo de muestreo, que describe como los datos están organizados en el buffer.

El entramado, realiza las siguientes tareas:

- Representa las coordenadas rectangulares de la imagen.
- Mantiene la imagen en memoria.
- Provee de mecanismos para crear múltiples subimágenes a partir de un solo buffer de los datos de la imagen.
- Provee de métodos para accesar a los pixeles específicos de una imagen.

La clase `BufferedImage` una serie de constantes para los diferentes tipos de imágenes (modelos de colores). La tabla 14.1 describe los diferentes tipos de imágenes.

Tabla 14.1 Tipos de Imagen de la clase `BufferedImage`.

Tipo	Descripción
TYPE_3BYTE_BGR	Representa una imagen con componentes RGB de 8 bits, con los colores azul, verde y rojo almacenados en 3 bytes.
TYPE_4BYTE_ABGR	Representa una imagen con componentes RGBA de 8 bits, con los colores azul, verde y rojo almacenados en 3 bytes y un byte para el valor de alfa. Los datos de los colores no están premultiplicados con el valor de alfa.
TYPE_4BYTE_ABGR_PRE	Representa una imagen con componentes RGBA de 8 bits, con los colores azul, verde y rojo almacenados en 3 bytes y un byte para el valor de alfa. Los datos de los colores están premultiplicados con el valor de alfa.
TYPE_BYTE_BINARY	Representa una imagen con pixeles opacos de 1, 2 o 4 bits empaquetados. Los valores de los pixeles representan indices a una tabla de colores.
TYPE_BYTE_GRAY	Representa una imagen con pixeles en tonos de grises. El valor del pixel es un tono de gris (0 a 255).
TYPE_BYTE_INDEXED	Representa una imagen con pixeles opacos de 1 byte. Los valores de los pixeles representan indices a una tabla de colores de 256 valores, 216 colores y el resto de tonos de grises.
TYPE_CUSTOM	Tipo de imagen no reconocido. Debe ser una imagen con formato especial.
TYPE_INT_ARGB	Representa una imagen con componentes RGBA de 8 bits, con los colores azul, verde y rojo y el valor de alfa empacados en enteros. Los datos de los colores no están premultiplicados con el valor de alfa.
TYPE_INT_ARGB_PRE	Representa una imagen con componentes RGBA de 8 bits, con los colores azul, verde y rojo y el valor de alfa empacados en enteros. Los datos de los colores están premultiplicados con el valor de alfa.
TYPE_INT_BGR	Representa una imagen con componentes RGBA de 8 bits, con los colores azul, verde y rojo empacados en enteros. No hay valor de alfa.
TYPE_INT_RGB	Representa una imagen con componentes RGBA de 8 bits, con los colores azul, verde y empacados en enteros. No hay valor de alfa.
TYPE_USHORT_555_RGB	Representa una imagen con componentes RGB de 5-5-5 bits para los colores azul, verde y rojo. No hay valor de alfa.
TYPE_USHORT_565_RGB	Representa una imagen con componentes RGB de 5-6-5 bits para los colores azul, verde y rojo. No hay valor de alfa.
TYPE_USHORT_GRAY	Representa una imagen con pixeles en tonos de grises.
TYPE_INT_ARGB	Representa una imagen con componentes RGBA de 8 bits, con los colores azul, verde y rojo y el valor de alfa empacados en enteros. Los datos de los colores no están premultiplicados con el valor de alfa.

La tabla 14.2 muestra algunos de los métodos de la clase `BufferedImage`.

Tabla 14.3 Métodos de la clase `BufferedImage`.

`public` **`BufferedImage`**`(int width, int height, int imageType)` Construye una imagen del tipo `BufferedImage` de alguno de los tipos de imagenes predefinidos.

`public int` **`getWidth`**`()`
`public int` **`getHeight`**`()`
 	
Regresan el ancho y la altura de la imagen del tipo `BufferedImage`, respectivamente.

`public Graphics2D` **`createGraphics`**`()` Crea un objeto del tipo `Graphics2D` que puede usarse para dibujar sobre esta imagen.
`public BufferedImage` **`getSubimage`**`(int x, int y, int w, int h)` Regresa una subimagen de esta imagen especificada por la región rectangular. La imagen regresada comparte el mismo arreglo de datos que la imagen original.

Dibujado de Imágenes

Al igual que con las gráficas, podemos dibujar una figura sobre cualquier componente de swing que sea desplegable. La clase `Graphics2D` dispone del método `drawImage()` cuya sintaxis se muestra en la tabla 14.4.

Tabla 14.4 Método para dibujar una Imagen `BufferedImage`.

`public abstract void` **`drawImage`**`(BufferedImage img, BufferedImageOp op,`
`int x, int y)`
 	
Despliega una imagen del tipo `BufferedImage`, que es filtrada con un filtro descrito por el objeto `op`, en las coordenadas (`x, y`).

El siguiente código muestra un método que permite dibujar una imagen

```
...

/**
 * Esta clase contiene métodos para cargar, guardar y procesar imagenes
 *
 * @author mdomitsu
 */
public class Control {
    ...
    // Imagen a ser desplegada
    private BufferedImage dbi;
    ...

    /**
     * Este metodo despliega la imagen del atributo dbi
     * del tipo BufferedImage sobre el panel lienzo
     * @param lienzo Panel sobre el que se despliega la imagen
     */
    public void despliegaimagen(JPanel lienzo) {
        // Si no hay una imagen para desplegar
        if (dbi == null) {
```

```
            JOptionPane.showMessageDialog(lienzo,
                    "No hay una imagen para desplegar");
            return;
        }
        // Obtiene el tamaño de la imagen
        int altoImagen = dbi.getHeight();
        int anchoImagen = dbi.getWidth();

        // Establece el tamaño del panel al tamaño de la imagen
        lienzo.setPreferredSize(new Dimension(altoImagen, anchoImagen));
        lienzo.revalidate();

        Graphics g = lienzo.getGraphics();
        Graphics2D g2 = (Graphics2D) g;

        // Dibuja la imagen sobre el panel
        g2.drawImage(dbi, null, 1, 1);
    }

    ...
}
```

La figura 14.3 muestra una imagen desplegada por el método anterior.

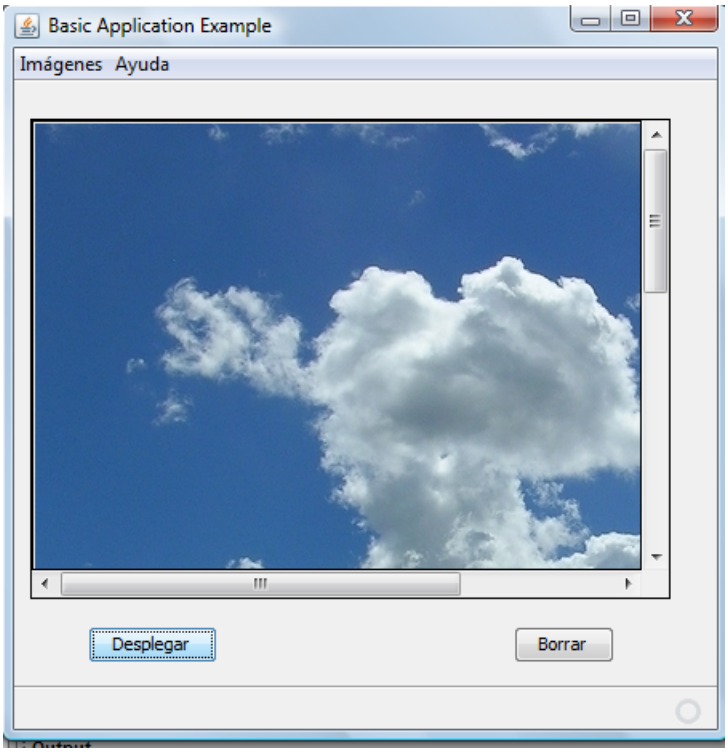

Figura 14.3 Despliegue de una Imagen

Creación de Imágenes

Podemos crear una nueva imagen (una imagen con todos los pixeles en cero) para posteriormente dibujar sobre ella. Para ello se sigue el siguiente procedimiento:

1. Crear un objeto del tipo `BufferedImage`.
2. Obtener del objeto del tipo `BufferedImage`, un objeto del tipo `Graphics2D`, para dibujar sobre la imagen.
3. Dibujar sobre la imagen usando los métodos del objeto del tipo `Graphics2D`.

El siguiente código muestra un método que permite crear una imagen nueva.

```
public class Control {
    ...
    // Imagen original
    private BufferedImage obi;

    /**
     * Este método crea un objeto de tipo BufferedImage del
     * tamaño del panel del parametro
     * @param lienzo Panel que establece el tamaño de la imagen.
     */
    public void nuevaImagen(JPanel lienzo) {
        // Obtiene el tamaño del panel
        int altoImagen = lienzo.getHeight();
        int anchoImagen = lienzo.getWidth();

        // Crea una imagen del tamaño del panel
        obi = new BufferedImage(anchoImagen, altoImagen,
                BufferedImage.TYPE_INT_RGB);

        Graphics2D g2 = obi.createGraphics();

        // Establece el color de fondo
        g2.setBackground(Color.white);
        g2.clearRect(0, 0, anchoImagen, altoImagen);

        // Hace que la referencia dbi apunte a obi
        dbi = obi;
    }

    ...
}
```

El siguiente código dibuja sobre una imagen (nueva o existente):

```
    /**
     * Este método dibuja una figura sobre la imagen obi
     * @param frame Ventana sobre la que se despliega el
     * mensaje de error
     */
    public void dibujaEnImagen(JFrame frame) {
        // Si no hay una imagen para dibujar
        if (obi == null) {
            JOptionPane.showMessageDialog(frame,
                    "No hay una imagen para dibujar");
            return;
        }

        // Obtiene el contexto de graficación de la imagen
        Graphics2D g2 = obi.createGraphics();

        // Establece el color de una trayectoria
        g2.setPaint(Color.black);
        // crea una trayectoria
        Path2D trayectoria = new Path2D.Double(Path2D.WIND_EVEN_ODD);
        trayectoria.moveTo(50, 50);
        trayectoria.lineTo(70, 44);
        trayectoria.curveTo(100, 10, 140, 80, 160, 80);
        trayectoria.lineTo(190, 40);
        trayectoria.lineTo(200, 56);
        trayectoria.quadTo(100, 150, 70, 60);
        trayectoria.closePath();

        // Dibuja la trayectoria sobre la imagen obi
        g2.draw(trayectoria);

        // Hace que la referencia dbi apunte a obi
        dbi = obi;
    }
```

Las figuras 14.4 y 14.5 muestran una figura sobre una nueva imagen y sobre una imagen existente, dibujada por el método anterior, respectivamente.

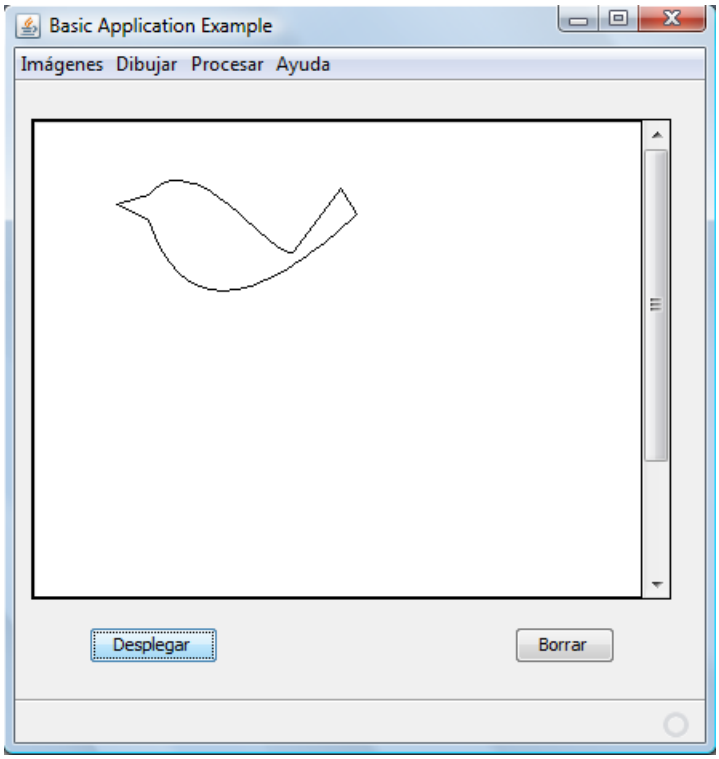

Figura 14.4 Figura Sobre una Nueva Imagen

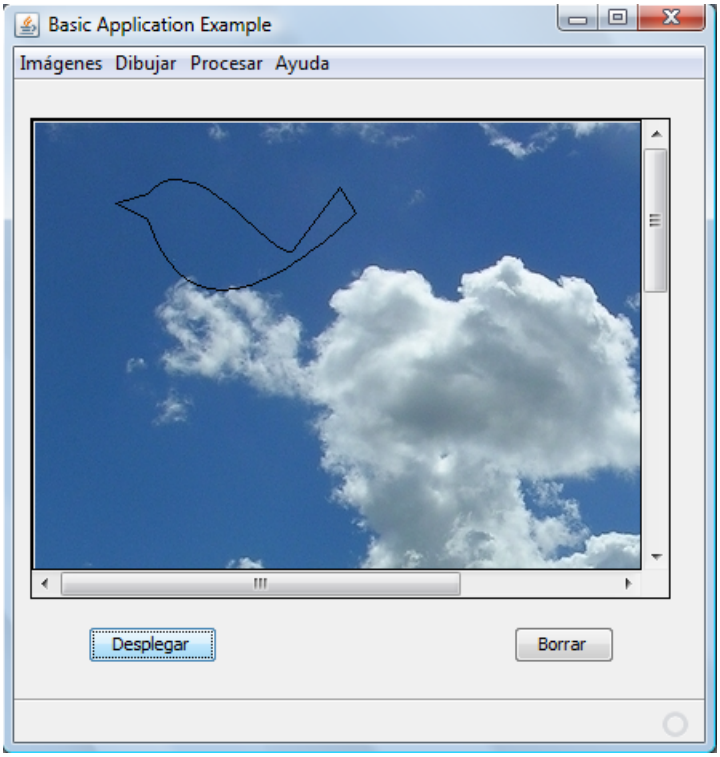

Figura 14.5 Figura Sobre una Imagen Existente

Entrada / Salida de Imágenes

Para cargar imágenes de archivos o guardar imágenes a archivos, Java nos proporciona la clase `ImageIO` que nos proporciona un conjunto de métodos estáticos para leer y escribir imágenes a archivos. El diagrama de clases de esa clase se muestra en la figura 14.6.

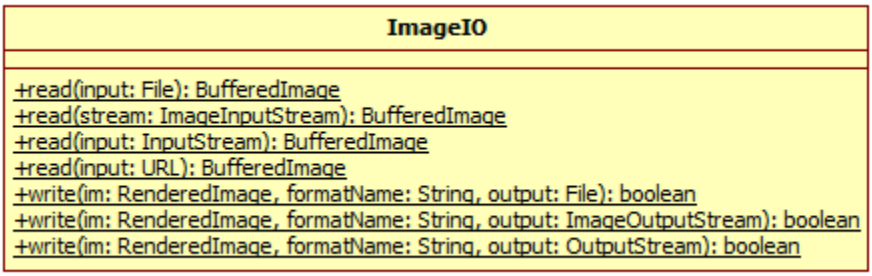

Figura 14.6 Clase ImageIO

La tabla 14.5 muestra los métodos de la clase `ImageIO`.

Tabla 14.5 Métodos de la Clase `ImageIO`

`public static BufferedImage` **`read`**`(File input) throws IOException` Regresa un objeto del tipo `BufferedImage` como resultado de decodificar el archivo dado por el parámetro, usando el objeto del tipo `ImageReader` seleccionado automaticamente de los registrados.

`public static BufferedImage` **`read`**`(ImageInputStream stream)`
`  throws IOException`
 	
Regresa un objeto del tipo `BufferedImage` como resultado de decodificar el objeto del tipo `ImageInputStream` dado por el parámetro, usando el objeto del tipo `ImageReader` seleccionado automaticamente de los registrados.

`public static BufferedImage` **`read`**`(InputStream input) throws IOException` Regresa un objeto del tipo `BufferedImage` como resultado de decodificar el objeto del tipo `InputStream` dado por el parámetro, usando el objeto del tipo `ImageReader` seleccionado automaticamente de los registrados.
`public static BufferedImage` **`read`**`(URL input) throws IOException` Regresa un objeto del tipo `BufferedImage` como resultado de decodificar el objeto del tipo `URL` dado por el parámetro, usando el objeto del tipo `ImageReader` seleccionado automaticamente de los registrados.

`public static boolean` **`write`**`(RenderedImage im, String formatName,`
`  File output) throws IOException`
 	
Escribe la imagen del parámetro `im` con el formato dado por `formatName` al archivo dado por `output`.

Tabla 14.5 Métodos de la Clase `ImageIO`. cont.

`public static boolean write(RenderedImage im, String formatName, ImageOutputStream output) throws IOException` Escribe la imagen del parámetro `im` con el formato dado por `formatName` al archivo, usando el objeto del tipo **`ImageOutputStream.`**
`public static boolean write(RenderedImage im, String formatName, ImageOutputStream output) throws IOException` Escribe la imagen del parámetro `im` con el formato dado por `formatName` al archivo, usando el objeto del tipo **`OutputStream.`**

Como ejemplo de entrada / salida de imágenes se tiene un programa con una interfaz de usuario gráfica. El programa permitirá leer y guardar imágenes así como procesar esas imágenes. La opción para leer una imagen de un archivo se muestra en la figura 14.7. Al seleccionar esa opción se despliega un cuadro de diálogo para seleccionar el nombre del archivo con la imagen a leer, figuras 14.8 y 14.9.

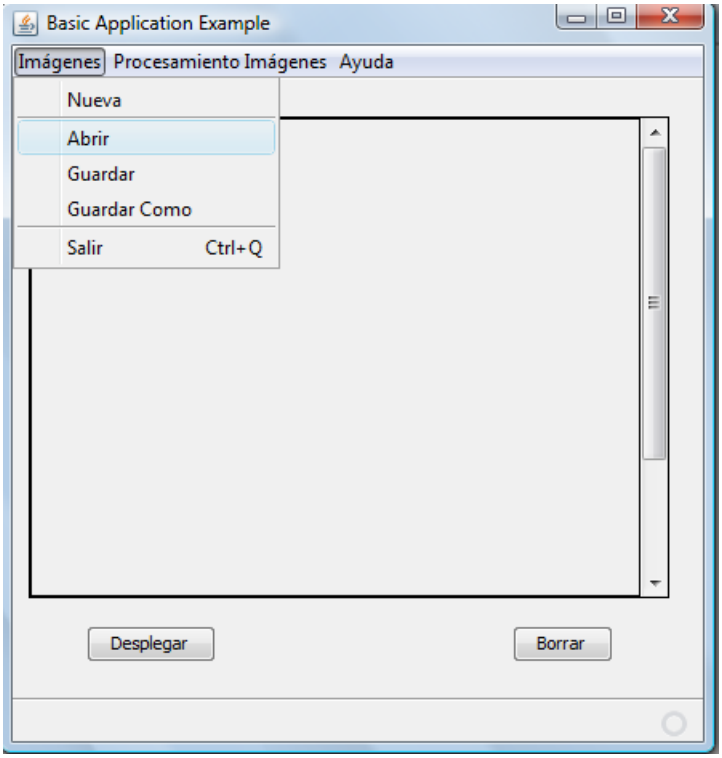

Figura 14.7

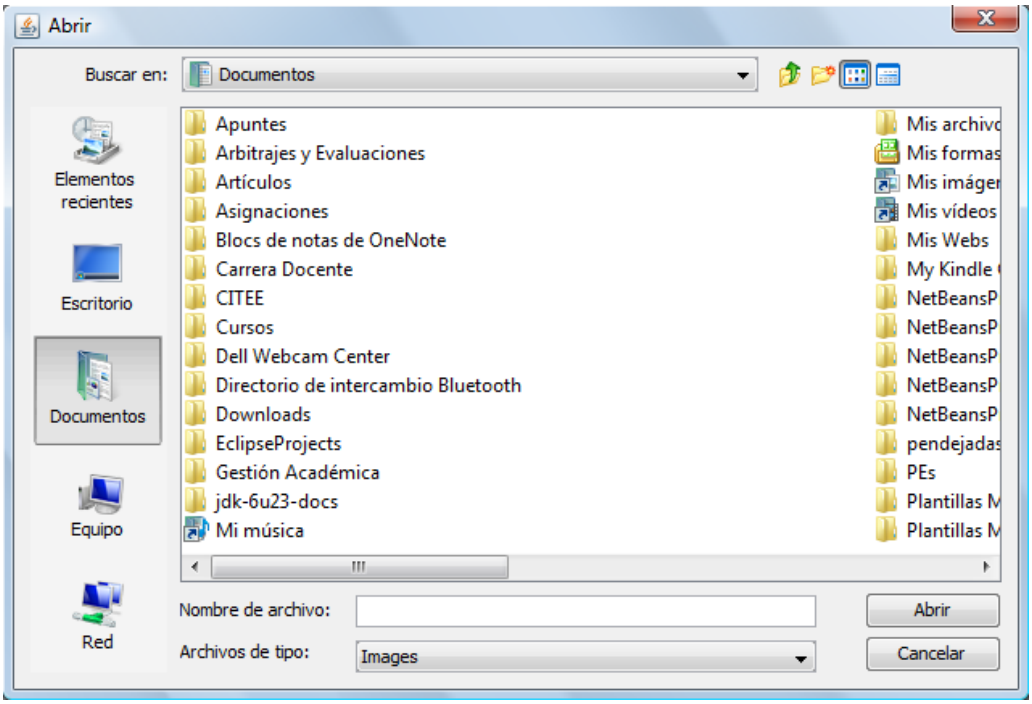

Figura 14.8

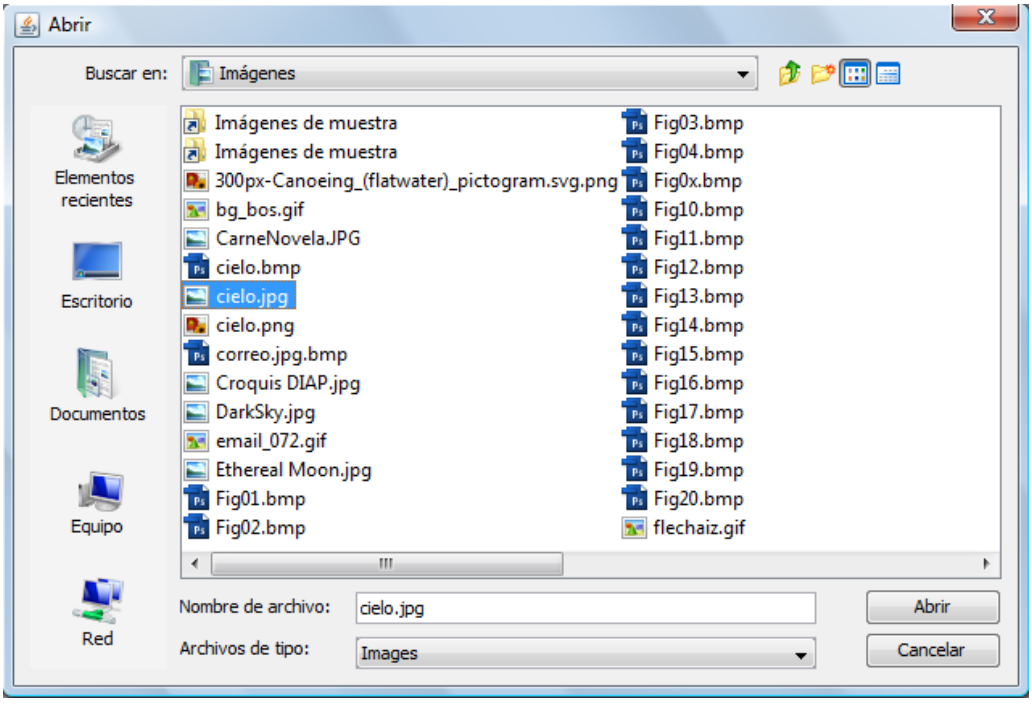

Figura 14.9

El código que permite leer una imagen de un archivo es el siguiente:

```
    /**
     * Este metodo lee una imagen bmp, gif, jpg, png y la
     * guarda en el atributo dbi del tipo BufferedImage
     * @param frame Ventana sobre la que se despliega el cuadro
     * de dialogo JFileChooser
     */
    public void leeImagen(JFrame frame) {
        JFileChooser fc = new JFileChooser();
        // Elimina el filtro *.*
        fc.setAcceptAllFileFilterUsed(false);
        // Crea el filtro para las extenciones validas
        FileNameExtensionFilter extFiltro = new FileNameExtensionFilter(
                "Images", "bmp", "gif", "jpg", "png");
        // Establece el filtro para las extenciones validas
        fc.setFileFilter(extFiltro);
        // Despliega el cuadro de dialogo para seleccionar la imagen a
        // abrir
        int returnVal = fc.showOpenDialog(frame);
        // Si se selecciono una imagen
        if (returnVal == JFileChooser.APPROVE_OPTION) {
            // Obtiene el objeto File de la imagen seleccionada
            file = fc.getSelectedFile();

            try {
                // lee la imagen y la guarda en el atributo obi
                // del tipo BufferedImage
                obi = ImageIO.read(file);

                // Hace que la referencia dbi apunte a obi
                dbi = obi;
            } catch (IOException e) {
                JOptionPane.showMessageDialog(frame,
                        "Error al cargar imagen");
                return;
            }
        }
    }
```

La opción para guardar una imagen de un archivo se muestra en la figura 14.7. Al seleccionar esa opción se despliega un cuadro de diálogo para seleccionar el nombre del archivo en el que se guardará la imagen, figuras 14.10.

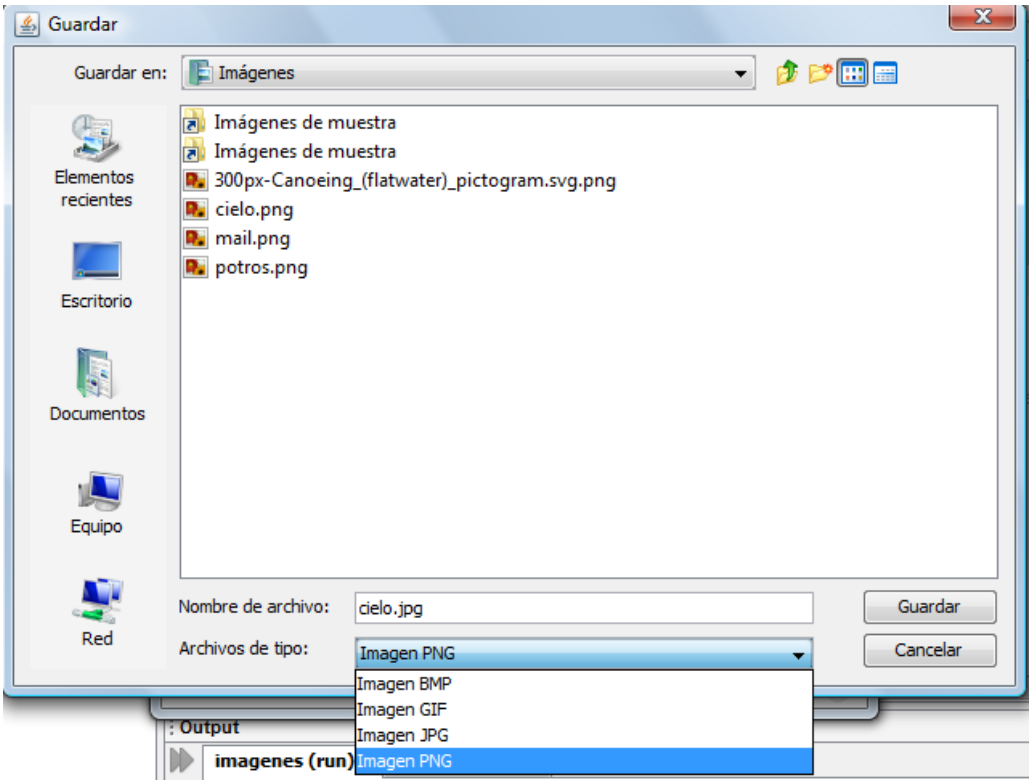

Figura 14.10

El código que permite guardar una imagen a un archivo es el siguiente:

```
    /**
     * Este metodo guarda la imagen bmp, gif, jpg, png del
     * atributo bi del tipo BufferedImage en un archivo
     * @param frame Ventana sobre la que se despliega el cuadro
     * de dialogo JFileChooser
     */
    public void GuardaImagenComo(JFrame frame) {
        File fileSel = null;

        JFileChooser fc = new JFileChooser();
        // Elimina el filtro *.*
        fc.setAcceptAllFileFilterUsed(false);
        // Agrega varios filtros de imagenes
        fc.addChoosableFileFilter(
                new FileNameExtensionFilter("Imagen BMP", "bmp"));
        fc.addChoosableFileFilter(
                new FileNameExtensionFilter("Imagen GIF", "gif"));
        fc.addChoosableFileFilter(
                new FileNameExtensionFilter("Imagen JPG", "jpg"));
        fc.addChoosableFileFilter(
                new FileNameExtensionFilter("Imagen PNG", "png"));
        //Establece el nombre inicial de la imagen
```

```
        fc.setSelectedFile(file);

        // Despliega cuadro de dialogo para obtener el nombre
        // del archivo en el que se va a guardar la imagen
        int returnVal = fc.showSaveDialog(frame);
        if (returnVal == JFileChooser.APPROVE_OPTION) {
            String nombreExt = null;

            // Obtiene el nombre del archivo seleccionado
            fileSel = fc.getSelectedFile();
            // Obtiene el nombre del filtro seleccionado
            FileNameExtensionFilter extFiltro =
               (FileNameExtensionFilter) fc.getFileFilter();
            // Obtiene la extension del nombre del filtro seleccionado
            String ext = extFiltro.getExtensions()[0];

            String path = fileSel.getPath();

            // Obtiene la extension del nombre del archivo seleccionado
            nombreExt = getExtension(fileSel);

            // Si el nombre seleccionado no corresponde a uno de imagen
            if(nombreExt != null && !esImageExtension(nombreExt)) {
                 JOptionPane.showMessageDialog(frame,
                        "No es un archivo de imagen");
                 return;
            }

            // Si no hay extension del nombre del archivo seleccionado
            if (nombreExt == null) {
                // Agregale la extension del nombre del filtro
                // seleccionado
                path +=  "." + ext;
                fileSel = new File(path);
                nombreExt = ext;
            }

            try {
                // Guarda la imagen
                ImageIO.write(bi, nombreExt, fileSel);
            } catch (IOException e) {
                 JOptionPane.showMessageDialog(frame,
                        "Error al guardar la imagen");
            }
        }
    }

    /**
     * Este metodo despliega la imagen del atributo bi
```

```
     * del tipo BufferedImage sobre el panel lienzo
     * @param lienzo Panel sobre el que se despliega la imagen
     */
    public void despliegaimagen(JPanel lienzo) {

        // Obtiene el tamaño de la imagen
        int altoImagen = bi.getHeight();
        int anchoImagen = bi.getWidth();

        lienzo.setPreferredSize(new Dimension(altoImagen, anchoImagen));
        lienzo.revalidate();

        Graphics g = lienzo.getGraphics();
        Graphics2D g2 = (Graphics2D) g;

        g2.drawImage(bi, null, 1, 1);
    }

    /**
     * Este metodo estatico borra el contenido del panel de
     * su parametro
     * un conjunto de lineas
     * @param lienzo Panel a borrar
     */
    public void borra(JPanel lienzo) {
        // Obtiene un objeto de tipo Graphics del panelLienzo
        Graphics g = lienzo.getGraphics();

        // Al invocar al metodo paint se borra su contenido
        lienzo.paint(g);
    }

    /**
     * Este metodo estatico obtiene la extension de un archivo
     * @param file Objeto de tipo File de la que se obtiene
     * la extension
     * @return Extension de un archivo
     */
    public static String getExtension(File file) {
        String ext = null;
        // Obtiene el nombre del archivo
        String s = file.getName();
        // busca el separador de la extension
        int pos = s.lastIndexOf('.');

        // Si hay un punto en el nombre y hay una
        // extension despues del punto
        if (pos > 0 && pos < s.length() - 1) {
            ext = s.substring(pos + 1).toLowerCase();
```

```
        }
        return ext;
    }

    /**
     * Este medodo determina si la extension del nombre de archivo
     * corresponte a una imagen
     * @param ext Extension del nombre de archivo
     * @return true si si la extension del nombre de archivo
     * corresponte a una imagen, false en caso contrario
     */
    public boolean esImageExtension(String ext) {
        String[] imagenesExt = {"bmp", "gif", "jpg", "png"};

        for(int i = 0; i < imagenesExt.length; i++) {
            if(ext.equals(imagenesExt[i])) return true;
        }

        return false;
    }
}
```

Procesamiento de Imágenes

El procesamiento de imágenes describe la forma de manipular matemáticamente las imágenes. Procesar una imagen consiste en calcular un nuevo coor para cada pixel de una imagen. Ese nuevo color puede depender del color actual del pixel, del color de los pixeles vecinos, de otros parámetros o una combinación de los anteriores.

El API 2D de Java nos permite una forma sencilla de procesar imágenes basada en la clase `BufferedImage` y un conjunto de operaciones representadas en la interfaz `BufferedImageOp`. Las clases que implementan esta interfaz saben como procesar una imagen del tipo `BufferedImage`, llamada imagen fuente para producir una nueva imagen llamada imagen destino. Este proceso se muestra en la figura 14.11.

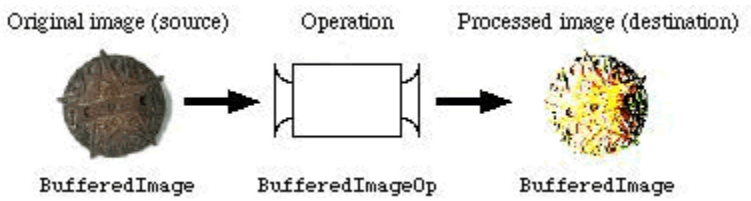

Figura 14.11.

Con la API 2D de Java, el procesamiento de imágenes es un procedimiento de dos pasos:

1. Instancíe la clase que represente la operación a realizar.
2. Invoque a su método `filter()` con la imagen como parámetro.

La interfaz `BufferedImageOp`, las clases que implementan los métodos de la interfaz y las clases que las soportan se muestran en la figura 14.12.

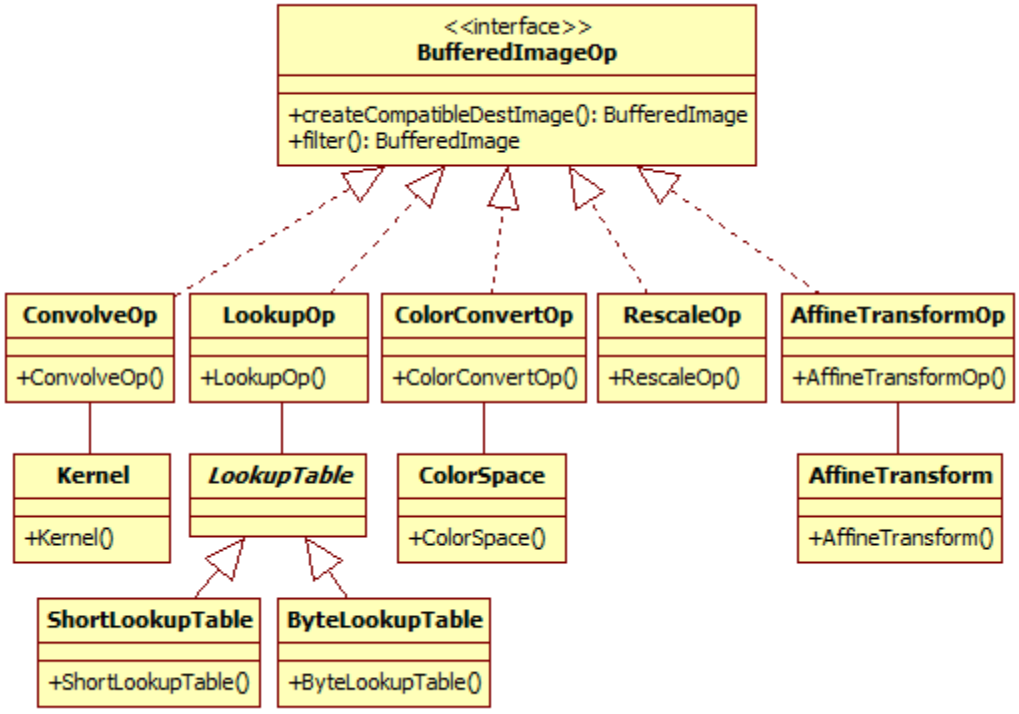

Figura 14.11.

Las operaciones que realizan las clases que implementan la interfaz `BufferedImageOp` se muestran en la tabla 14.6.

Tabla 14.6. Operaciones de Procesamiento de Imágenes

Clase	Operación
`ConvolveOp`	Convolución: Suavizado, intensificado, detección de orillas.
`LookupOp`	Tabla de búsquedas: Posterizar, binarizar.
`ColorConvertOp`	Conversión de colores: Convertir a tonos de grises.
`RescaleOp`	Reescalar colores: Aclarar, oscurecer.
`AffineTransformOp`	Transformar: Escalar, rotar.

Convolución

Una operación de convolución permite combinar el color de un pixel origen con los pixeles de sus vecinos para producir un pixel destino. Esa combinación se especifica mediante un operador lineal llamado kernel (núcleo) que determina la proporción de cada pixel de origena usarce para calcular el pixel destino. Un kernel es una matriz donde el centro de la matriz representa el pixel origen y los otros elementos sus vecinos. El color destino se calcula multiplicando cada color pixel por su correspondiente coeficiente del kernel y sumándolos, figura 14.12.

La clase `Kernel` representa un kernel. Para construir un kernel podemos usar el constructor de la tabla 14.7.

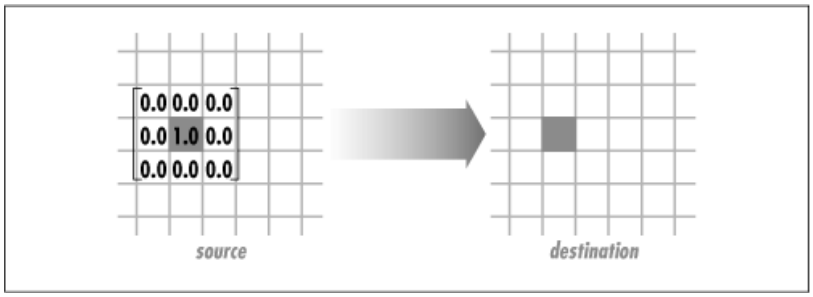

Figura 14.12. Convolución de un Solo Pixel.

Tabla 14.7 Constructor de la clase `Kernel`.

`public` **`Kernel`**`(int width, int height, float[] data)` Construye un kernel a partir del arreglo de números flotantes dados por el parámetro `data`. Las dimensiones del kernel están dados por los parámetros `width, height`.

En un kernel, la suma de todos sus elementos debe ser 1.0. Si es mayor la imagen resultante será más clara. Si es menor, será más oscura.

La operación de convolución está representada por la clase `ConvolveOp`. Los métodos de la clase `ConvolveOp` se muestran en la tabla 14.8.

Tabla 14.8 Métodos de la clase `ConvolveOp`.

`public` **`ConvolveOp`**`(Kernel kernel)`
`public` **`ConvolveOp`**`(Kernel kernel, int edgeCondition, RenderingHints hints)`
 	
Construye un objeto del tipo `ConvolveOp` a partir de un kernel o un kernel, una constante que represente una condiciónde orilla y una sugerencia de despliegue, la cual puede ser nula.

Tabla 14.8 Métodos de la clase `ConvolveOp`. Cont.

`public final BufferedImage` **`filter`**`(BufferedImage src, BufferedImage dst)` Realiza la operación de convolución de la imagen bufereada dada por el parámetro `src`. Cada componente de la imagen será convolucionado (incluyendo la componente alfa, si están presente). Si el modelo de color de la imagen fuente no es el mismo del de la imagen destino, el modelo de color será convertido al de la imagen destino. Si el parámetro `dst` es null, se creará una nueva imagen con el modelo de color de la imagen fuente.

El parámetro `edgeCondition` establece la forma en que se calculan los colores de los pixeles de la orilla de la imagen destino.

Tabla 14.9. Tipos de Condiciones de Orilla

Tipo	Descripción
`EDGE_ZERO_FILL`	Los pixeles en las orillas de la imagen destino se establecen a cero. Este es el valor por ausencia.
`EDGE_NO_OP`	Los pixeles en las orillas de la imagen fuente son copiados a sus correspondientes pixeles en la imagen destino sin modificación.

Suavizado de Imágenes

Una de las operaciones de convolución es el suavizado. En esta operación el color de un pixel destino es el promedio de su correspondiente pixel origen con sus pixeles vecinos. Este tipo de filtrado es útil para reducir el ruido de las imágenes.

El siguiente código muestra la implementación de un filtro suavizador.

```
    /**
     * Este metodo suaviza una imagen
     * @param frame Ventana sobre la que se despliega el mensaje de error
     */
    public void suaviza(JFrame frame) {
        // verifica que haya una imagen lista para ser procesada
        if(!preparaImagenProcesar(frame))
            return;

        // Define un kernel suavizador
        float ninth = 1.0f / 9.0f;
        float[] blurKernel = {
            ninth, ninth, ninth,
            ninth, ninth, ninth,
            ninth, ninth, ninth
        };

        // crea una operación de convolución a partir del kernel
        ConvolveOp op = new ConvolveOp(new Kernel(3, 3, blurKernel));
```

```
        // Filtra la imagen usando la operacion
        pbi = op.filter(pbi, null);

        // Hace que la referencia dbi apunte a obi
        dbi = pbi;
    }

    /**
     * Este metodo verifica que haya una imagen lista para ser procesada
     * @param frame Ventana sobre la que se despliega el mensaje de error
     */
    private boolean preparaImagenProcesar(JFrame frame) {
        // Si no hay se ha creado una imagen vacia o se ha
        // cargado una imagen desde un archivo
        if (obi == null) {
            JOptionPane.showMessageDialog(frame,
                    "No hay una imagen para procesar");
            return false;
        }

        // Si no hay una imagen a procesar
        if (pbi == null) {
            // Crea una copia de la imagen original y la asigna a la
            // imagen a procesar
            pbi = obi.getSubimage(0, 0, obi.getWidth(), obi.getHeight());

            // Obtiene el tamaño de la imagen
            int altoImagen = pbi.getHeight();
            int anchoImagen = pbi.getWidth();

            // Convierte la imagen a una imagen RGB normalizada
            pbi = new BufferedImage(anchoImagen, altoImagen,
                                    BufferedImage.TYPE_INT_RGB);
            Graphics2D g2 = pbi.createGraphics();
            g2.drawImage(dbi, null, 0, 0);
        }

        return true;
    }
```

La figura 14.13 muestra la imagen original y la imagen suavizada.

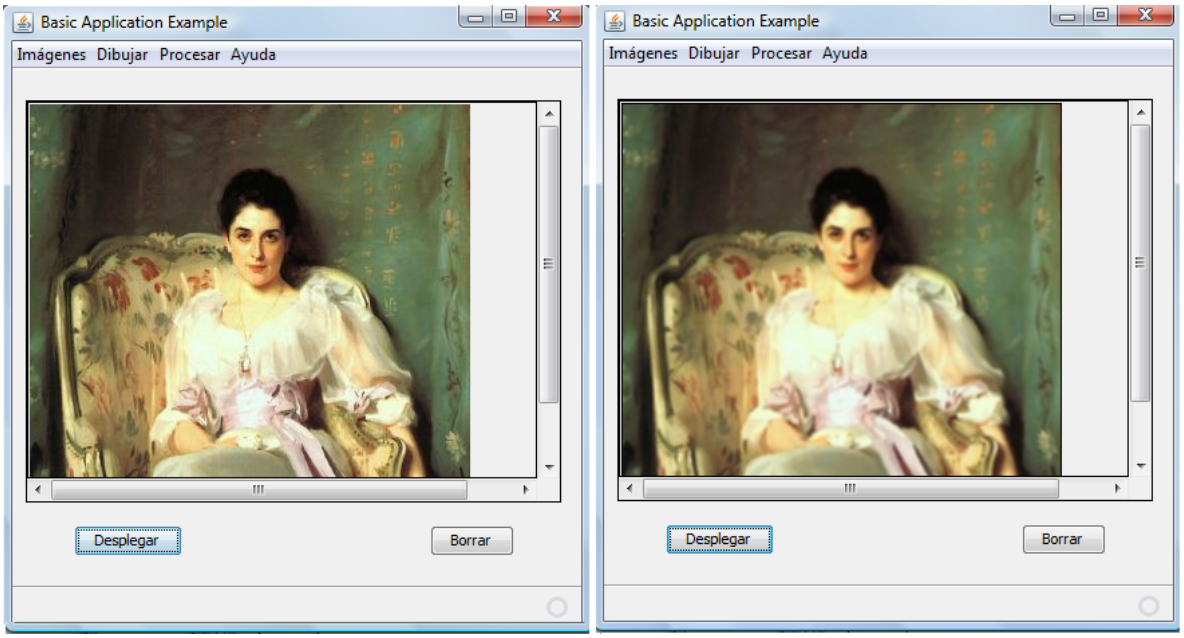

Figura 14.13. Filtro Suavizador. Izquierda: Imagen original, derecha: Imagen Suavizada.

Intensificado de Imágenes

Otra de las operaciones de convolución es el intensificado. En esta operación se busca resaltar el cambio en los colores de los pixeles, manteniendo los colores originales.

El siguiente código muestra la implementación de un filtro intensificador.

```
    /**
     * Este metodo intensifica una imagen
     * @param frame Ventana sobre la que se despliega el mensaje de error
     */
    public void intensifica(JFrame frame) {
        // verifica que haya una imagen lista para ser procesada
        if(!preparaImagenProcesar(frame))
            return;

        // Define un kernel intensificador
        float[] sharpKernel = {
            0.0f, -1.0f, 0.0f,
            -1.0f, 5.0f, -1.0f,
            0.0f, -1.0f, 0.0f
        };

        // crea una operación de convolución a partir del kernel
        ConvolveOp op = new ConvolveOp(new Kernel(3, 3, sharpKernel));
```

```
        // Filtra la imagen usando la operacion
        pbi = op.filter(pbi, null);

        // Hace que la referencia dbi apunte a obi
        dbi = pbi;
    }
```

La figura 14.14 muestra la imagen original y la imagen intensificada.

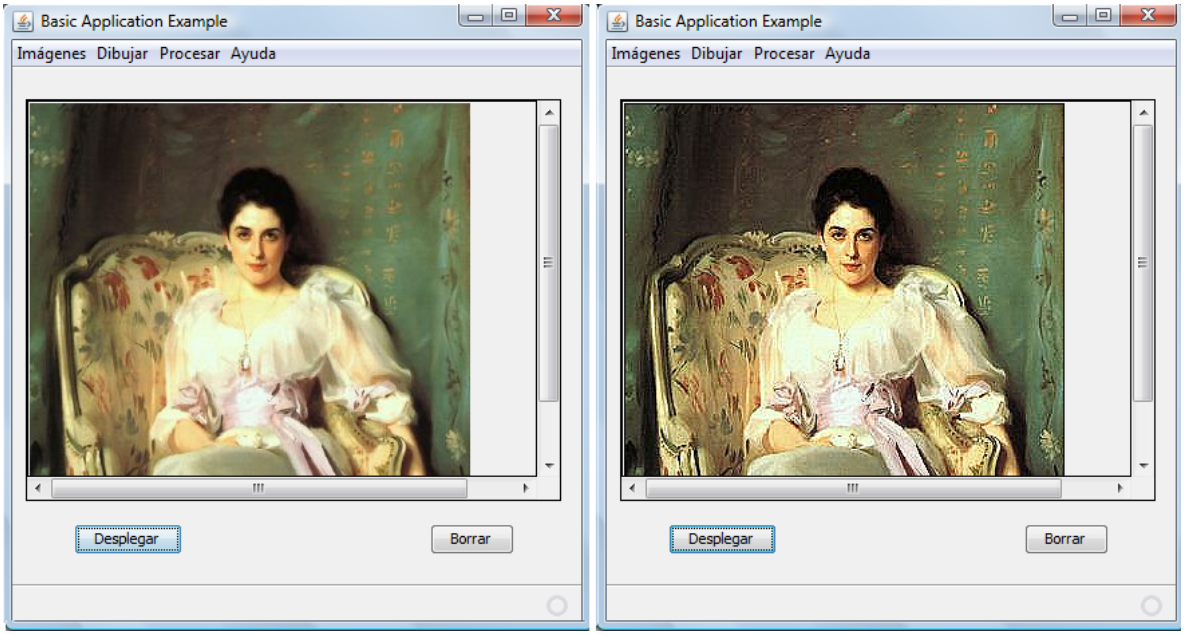

Figura 14.14. Filtro Intensificador. Izquierda: Imagen original, derecha: Imagen Intensificada.

Detección de Orillas

Otra operación de convolución muy empleada es la de detección de orillas. En esta operación se busca resaltar el cambio brusco en los colores de los pixeles.

El siguiente código muestra la implementación de un filtro detector de orillas.

```
    /**
     * Este metodo detecta las orillas de una imagen
     * @param frame Ventana sobre la que se despliega el mensaje de error
     */
    public void detectorOrillas(JFrame frame) {
        // verifica que haya una imagen lista para ser procesada
        if(!preparaImagenProcesar(frame))
            return;
```

```
        // Define un kernel detector de orillas
        float[] edgeKernel = {
            0.0f, -1.0f, 0.0f,
            -1.0f, 4.0f, -1.0f,
            0.0f, -1.0f, 0.0f
        };

        // crea una operación de convolución a partir del kernel
        ConvolveOp op = new ConvolveOp(new Kernel(3, 3, edgeKernel));

        // Filtra la imagen usando la operacion
        pbi = op.filter(pbi, null);

        // Hace que la referencia dbi apunte a obi
        dbi = pbi;
    }
```

La figura 14.15 muestra la imagen original y la imagen con las orillas detectadas.

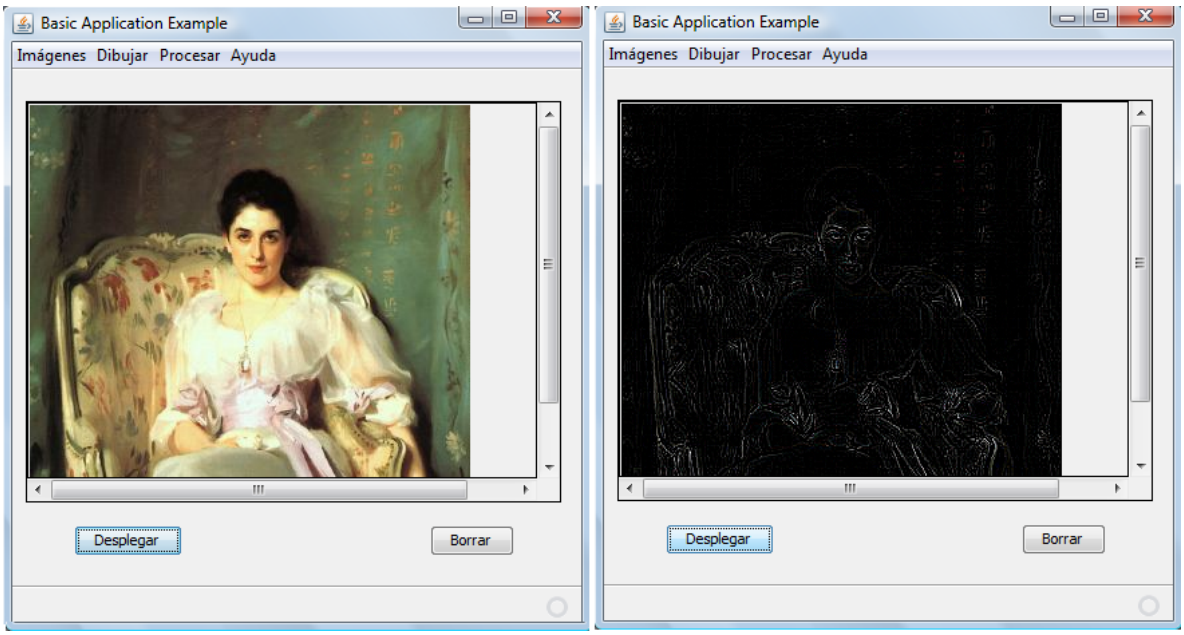

Figura 14.15. Filtro Detector de Orillas. Izquierda: Imagen original, derecha: Imagen Filtrada.

Tablas de Búsqueda

Otra operación sobre las imágenes utiliza una tabla de búsqueda. En esta operación, los pixeles de la imagen fuente se transforman a los pixeles de la imagen destino mediante el uso de una tabla. Tres tablas con 256 entradas cada

una son suficientes para transformar una imagen de color origen a una imagen de color destino. El valor de cada color de un pixel en la imagen origen se utiliza como índice en su correspondiente tabla para obtener el valor del pixel de la imagen destino.

Se puede utilizar una tabla para cada color, o la misma tabla para los tres colores. Cada tabla está encapsulada por la clase abstracta `LookupTable` y sus implementaciones `ByteLookupTable` y `ShortLookupTable`, que contienen arreglos de bytes y enteros cortos, respectivamente. Para construir tablas de búsqueda se tienen los constructores de las tablas 14.10.

Tabla 14.10 Constructores de las clases `ByteLookupTable` y `ShortLookupTable`

`public` **`ByteLookupTable`**`(int offset, byte[] data)` Construye un objeto del tipo `ByteLookupTable` del arreglo de bytes del parámetro `data` que representa una tabla de búsqueda que se aplicará a todas las bandas. El valor del parámetro `offset` se le quitará a los valores de entrada antes de usarlos como índices al arreglo.
`public` **`ByteLookupTable`**`(int offset, byte[][] data)` Construye un objeto del tipo `ByteLookupTable` del arreglo de arreglos de bytes del parámetro `data` que representan una tabla de búsqueda para cada banda. El valor del parámetro `offset` se le quitará a los valores de entrada antes de usarlos como índices a los arreglos. El número de bandas es es la longitud del parámetro `data`.
`public` **`ShortLookupTable`**`(int offset, short[] data)` Construye un objeto del tipo `ShortLookupTable` del arreglo de enteros cortos del parámetro `data` que representa una tabla de búsqueda que se aplicará a todas las bandas. El valor del parámetro `offset` se le quitará a los valores de entrada antes de usarlos como índices al arreglo.
`public` **`ShortLookupTable`**`(int offset, short[][] data)` Construye un objeto del tipo `ShortLookupTable` del arreglo de arreglos de bytes del parámetro `data` que representan una tabla de búsqueda para cada banda. El valor del parámetro `offset` se le quitará a los valores de entrada antes de usarlos como índices a los arreglos. El número de bandas es es la longitud del parámetro `data`.

La operación usando tablas de búsqueda está representada por la clase `LookUpOp`. Los métodos de la clase `LookUpOp` se muestran en la tabla 14.11.

Tabla 14.11 Métodos de la clase LookUpOp.

`public` **`LookUpOp`**`(`**`ByteLookupTable`** `table, RenderingHints hints)`
`public` **`LookUpOp`**`(`**`ShortLookupTable`** `table, RenderingHints hints)`
 	
Construye un objeto del tipo `LookUpOp` a partir de una tabla de búsqueda y una sugerencia de despliegue, la cual puede ser nula.

`public final BufferedImage` **`filter`**`(BufferedImage src, BufferedImage dst)` Realiza la operación de filtrado usando una tabla de búsqueda de de la imagen bufereada dada por el parámetro `src`. Si el modelo de color de la imagen fuente no es el mismo del de la imagen destino, el modelo de color será convertido al de la imagen destino. Si el parámetro `dst` es null, se creará una nueva imagen con el modelo de color de la imagen fuente.

Negativo de Imágenes

Una de las operaciones usando tablas de búsqueda es la inversión de colores. En esta operación el color de un pixel destino se obtiene restándole a 255, el valor de su correspondiente pixel origen. La imagen resultante luce como el negativo de color de una película convencional.

El siguiente código muestra la implementación de un filtro inversor de colores.

```
    /**
     * Este metodo invierte los colores de los pixeles de una image,
     * haciendo los pixeles claros oscuros y viceversa.
     * @param frame Ventana sobre la que se despliega el mensaje de error
     */
    public void invertir(JFrame frame) {
        // verifica que haya una imagen lista para ser procesada
        if(!preparaImagenProcesar(frame))
            return;

        // Define una tabla de busqueda para invertir los tres canales.
        // Se aplica la misma tabla para los tres canales
        short[] invert = new short[256];
        for (int i = 0; i < 256; i++) {
            invert[i] = (short) (255 - i);
        }
        LookupTable table = new ShortLookupTable(0, invert);

        // crea una operación de busqueda a partir de la tabla
        LookupOp op = new LookupOp(table, null);

        // Filtra la imagen usando la operacion
        pbi = op.filter(pbi, null);

        // Hace que la referencia dbi apunte a obi
        dbi = pbi;
    }
```

La figura 14.16 muestra la imagen original y la imagen con los colores invertidos.

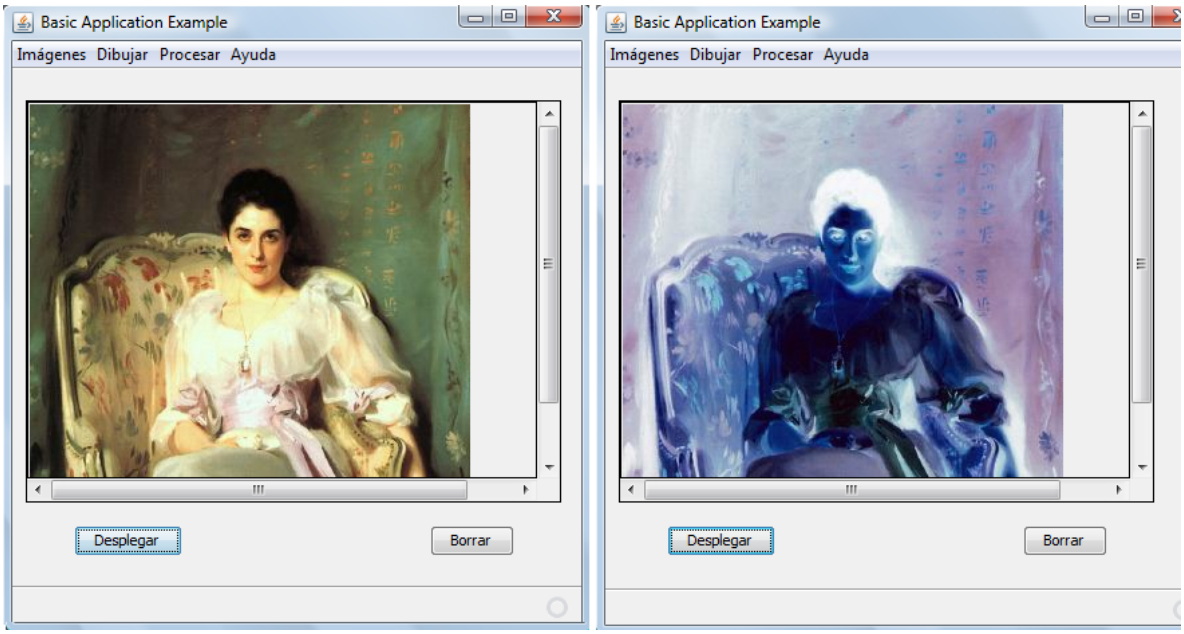

Figura 14.16. Inversión de colores. Izquierda: Imagen original, derecha: Imagen Negativa.

En el siguiente código se muestra una tabla de búsqueda en la que se tiene una tabla para cada canal. La operación elimina la componente verde de la imagen haciendo ceros todos los elementos del arreglo asociado a la banda verde.

```
    /**
     * Este metodo elimina la componente verde de una imagen.
     * @param frame Ventana sobre la que se despliega el mensaje de error
     */
    public void eliminarVerdes(JFrame frame) {
        // verifica que haya una imagen lista para ser procesada
        if(!preparaImagenProcesar(frame))
            return;

        // Define una tabla de busqueda para eliminar la componente
        // verde de la imagen. Se aplican tabla diferentes para los
        // canales
        short[] zero = new short[256];
        short[] straight = new short[256];
        for (int i = 0; i < 256; i++) {
            zero[i] = (short) 0;
            straight[i] = (short) i;
        }
        short[][] greenRemove = {straight, zero, straight};
        LookupTable table = new ShortLookupTable(0, greenRemove);

        // crea una operación de busqueda a partir de la tabla
        LookupOp op = new LookupOp(table, null);

        // Filtra la imagen usando la operacion
        pbi = op.filter(pbi, null);
```

```
        // Hace que la referencia dbi apunte a obi
        dbi = pbi;
    }
```

La figura 14.17 muestra la imagen original y la imagen con los colores verdes removidos.

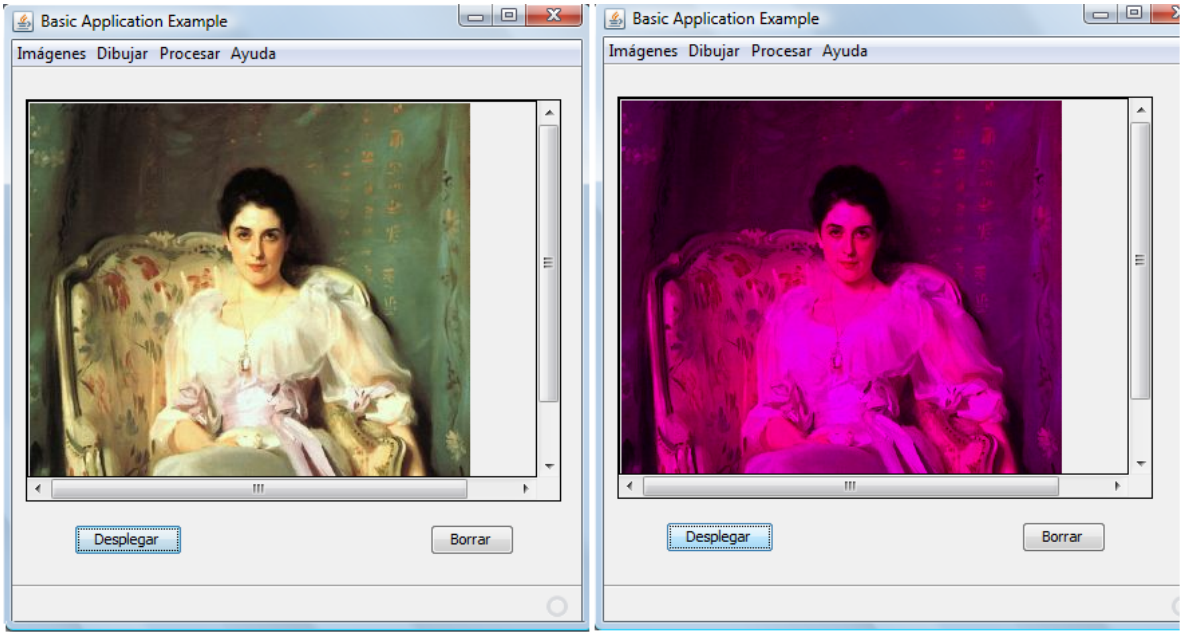

Figura 14.17. Remoción de colores. Izquierda: Imagen original, derecha: Imagen con los colores verdes removidos.

Aclarado de Imágenes Usando un Mapeado Lineal

Otro tipo de operación usando tablas de búsqueda es la que nos permite aclarar o aumentar la brillantez de una imagen usando un mapeado lineal. En esta operación el rango completo de valores de un color (0 a 255) se mapean a un rango menor (digamos de 128 a 255). La figura 14.18 muestra esa relación.

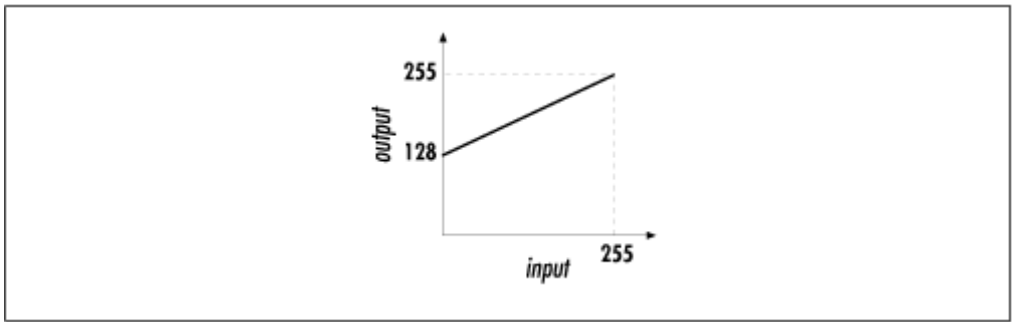

Figura 14.18. Abrillantando una imagen con un mapeo lineal.

El siguiente código muestra la implementación de un filtro aclarador lineal.

```
    /**
     * Este metodo aumenta la brillantez de una imagen
     * utilizando una transformacion lineal.
     * @param frame Ventana sobre la que se despliega el mensaje de error
     */
    public void aclaradoLineal(JFrame frame) {
        // verifica que haya una imagen lista para ser procesada
        if(!preparaImagenProcesar(frame))
            return;

        // Define una tabla de busqueda para aumentar la
        // brillantez de una imagen usando una transformacion
        // lineal
        short[] brighten = new short[256];
        for (int i = 0; i < 256; i++) {
            brighten[i] = (short) (128 + i / 2);
        }
        LookupTable table = new ShortLookupTable(0, brighten);

        // crea una operación de busqueda a partir de la tabla
        LookupOp op = new LookupOp(table, null);

        // Filtra la imagen usando la operacion
        pbi = op.filter(pbi, null);

        // Hace que la referencia dbi apunte a obi
        dbi = pbi;
    }
```

La figura 14.19 muestra la imagen original y la imagen aclarada usando un mapeo lineal.

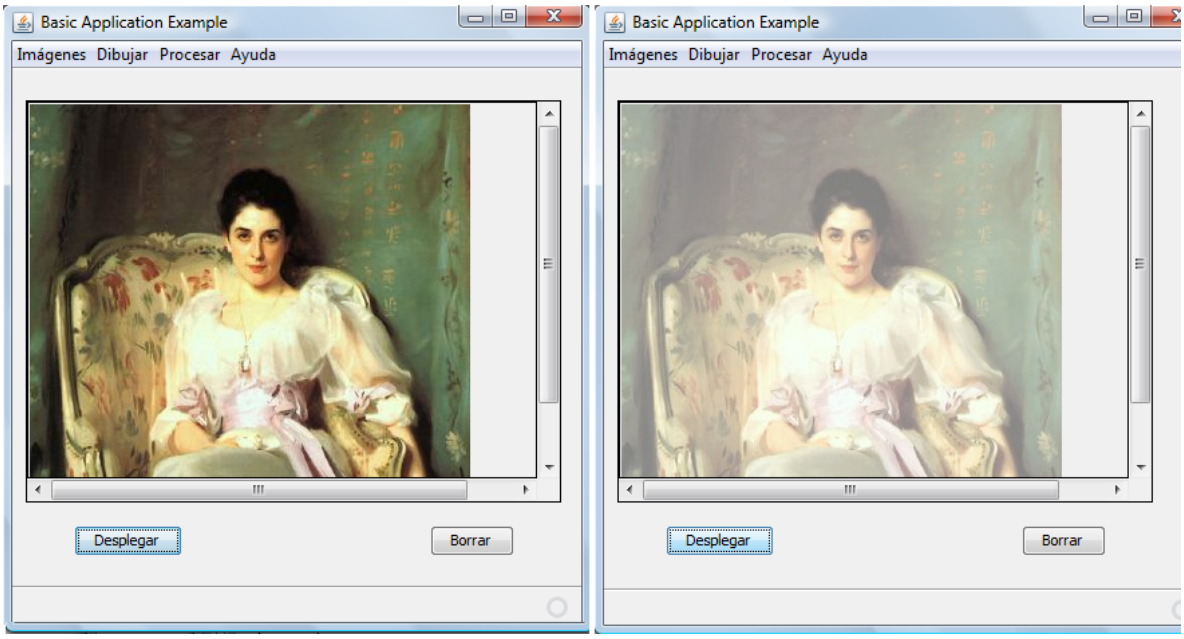

Figura 14.19. Abrillantando una imagen con un mapeo lineal. Izquierda: Imagen original, derecha: Imagen aclarada.

Aclarado de Imágenes Usando un Mapeado Raíz Cuadrática

El abrillantado de imágenes usando un mapeo lineal, produce imágenes que aparecen deslavadas. Otro tipo de operación de abrillantado, utiliza una función raíz cuadrada, como se muestra en la figura 14.20. El efecto producido es el de aclarar más el rango medio de los valores de entrada que los valores en el tope y en el fondo.

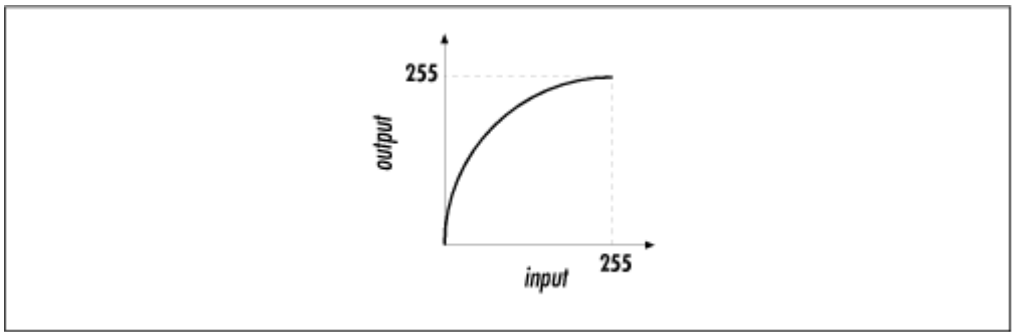

Figura 14.20. Abrillantando una imagen con un mapeo raíz cuadrática.

El siguiente código muestra la implementación de un filtro aclarador raíz cuadrática.

```
    /**
     * Este metodo aumenta la brillantez de una imagen
     * utilizando una transformacion raiz cuadratica.
     * @param frame Ventana sobre la que se despliega el mensaje de error
     */
    public void aclaradoRaiz(JFrame frame) {
        // verifica que haya una imagen lista para ser procesada
        if(!preparaImagenProcesar(frame))
            return;

        // Define una tabla de busqueda para aumentar la
        // brillantez de una imagen usando una transformacion
        // raiz cuadratica.
        short[] rootBrighten = new short[256];
        for (int i = 0; i < 256; i++) {
            rootBrighten[i] = (short) (Math.sqrt((double) i / 255.0)
                                * 255.0);
        }
        LookupTable table = new ShortLookupTable(0, rootBrighten);

        // crea una operación de busqueda a partir de la tabla
        LookupOp op = new LookupOp(table, null);

        // Filtra la imagen usando la operacion
        pbi = op.filter(pbi, null);

        // Hace que la referencia dbi apunte a obi
        dbi = pbi;
    }
```

La figura 14.21 muestra la imagen original y la imagen aclarada usando un mapeo raíz cuadrática.

Posterización

El efecto de posterización es la reducción del número de colores usados para desplegar una imagen. Para ello podemos usar una tabla de búsqueda que mapea valores de entrada a un conjunto pequeño de valores de salida.

El siguiente código muestra la implementación de un filtro posterizador.

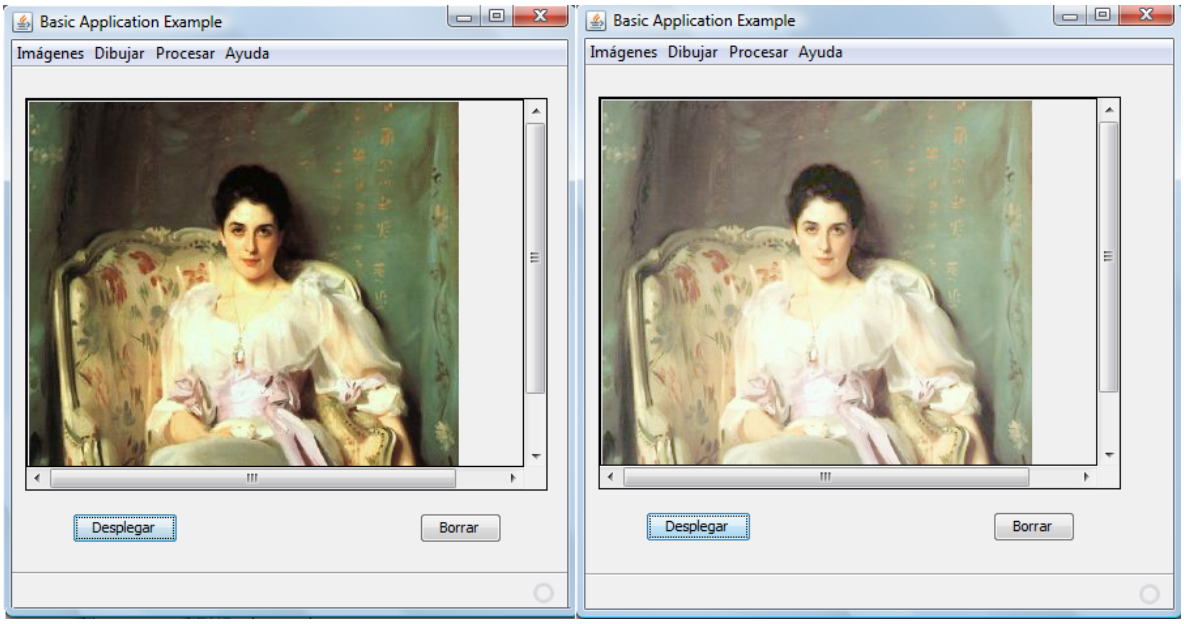

Figura 14.21. Abrillantando una imagen con un mapeo raíz cuadrática. Izquierda: Imagen original, derecha: Imagen aclarada.

```
    /**
     * Este metodo reduce el maximo numero de colores de
     * 2^24 = 16777216 a 2^9 = 256.
     * @param frame Ventana sobre la que se despliega el mensaje de error
     */
    public void posterizar(JFrame frame) {
        // verifica que haya una imagen lista para ser procesada
        if(!preparaImagenProcesar(frame))
            return;

        // Define una tabla de busqueda para reducir el maximo
        // numero de colores de 2^24 = 16777216 a 2^9 = 256.
        short[] posterize = new short[256];
        for (int i = 0; i < 256; i++) {
            posterize[i] = (short) (i - (i % 32));
        }
        LookupTable table = new ShortLookupTable(0, posterize);

        // crea una operación de busqueda a partir de la tabla
        LookupOp op = new LookupOp(table, null);

        // Filtra la imagen usando la operacion
        pbi = op.filter(pbi, null);

        // Hace que la referencia dbi apunte a obi
        dbi = pbi;
    }
```

La figura 14.22 muestra la imagen original y la imagen posterizada.

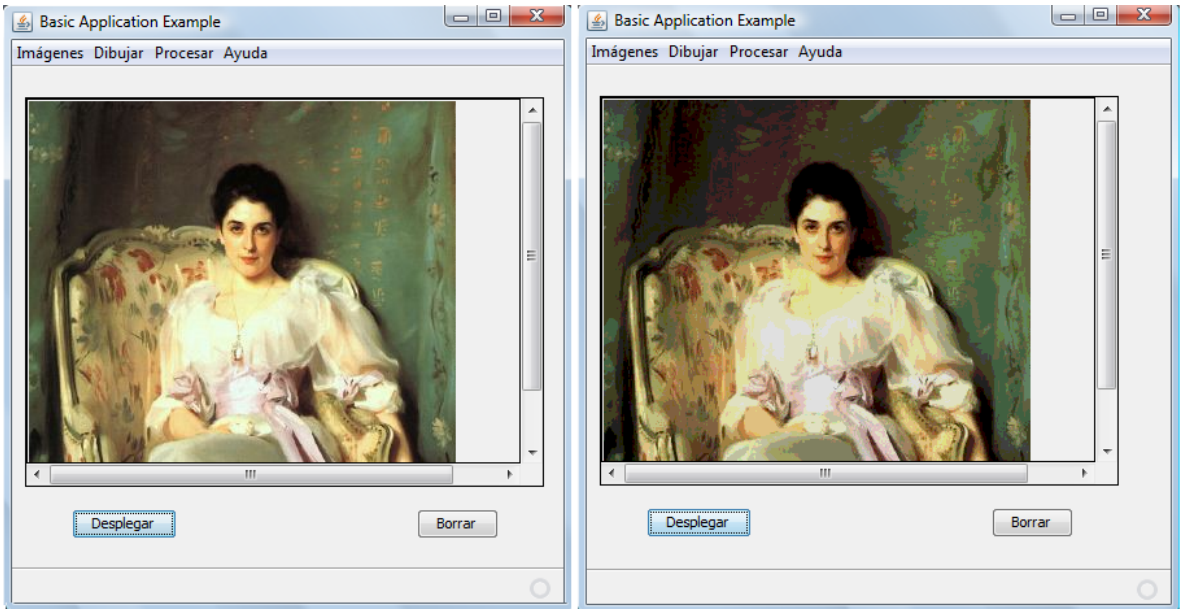

Figura 14.22. Posterizado de una imagen Izquierda: Imagen original, derecha: Imagen posterizada.

Binarizado

En esta operación, se usan tres valores: un valor límite, un valor mínimo y un valor máximo para controlar de los colores de cada pixel de la imagen. A los valores de un color por debajo del límite se les asigna el valor mínimo. A los valores de un color por encima del límite se les asigna el valor máximo.

El siguiente código muestra la implementación de un filtro posterizador con un valor mínimo de 0, un valor máximo de 255. Por lo que la imagen resultante contendrá sólo los siguientes 8 colores:

1. negro (rojo = 0, verde = 0, azul = 0)
2. blanco (rojo = 255, verde = 255, azul = 255)
3. rojo (rojo = 255, verde = 0, azul = 0)
4. verde (rojo = 0, verde = 255, azul = 0)
5. azul (rojo = 0, verde = 0, azul = 255)
6. amarillo (rojo = 255, verde = 255, azul = 0)
7. magenta (rojo = 255, verde = 0, azul = 255)
8. cian (rojo = 0, verde = 255, azul = 255)

```
    /**
     * Este metodo binariza una imagen reduciendo cada banda
     * de color a dos posibles valores 0 o 1, reduciendo el
     * numero de colores a 2^3 = 8.
     * @param frame Ventana sobre la que se despliega el mensaje de error
     */
    public void binarizar(JFrame frame) {
        // verifica que haya una imagen lista para ser procesada
        if(!preparaImagenProcesar(frame))
            return;

        // Define una tabla de busqueda para binarizar una imagen
        short[] threshold = new short[256];
        for (int i = 0; i < 256; i++) {
            threshold[i] = (i < 128) ? (short) 0 : (short) 255;
        }
        LookupTable table = new ShortLookupTable(0, threshold);

        // crea una operación de busqueda a partir de la tabla
        LookupOp op = new LookupOp(table, null);

        // Filtra la imagen usando la operacion
        pbi = op.filter(pbi, null);

        // Hace que la referencia dbi apunte a obi
        dbi = pbi;
    }
```

La figura 14.23 muestra la imagen original y la imagen binarizada.

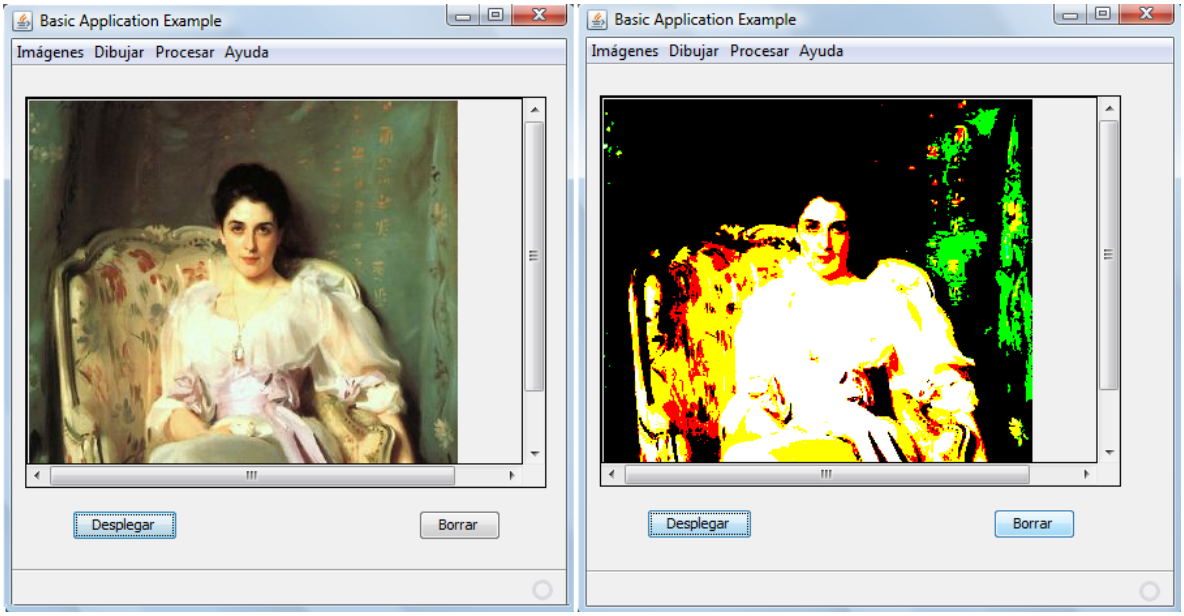

Figura 14.23. Binarizado de una imagen Izquierda: Imagen original, derecha: Imagen binarizada.

Conversión del Espacio de Color

Esta operación se utiliza para convertir los colores de un espacio de color a otro.

Los espacios de color estan representados por la clase `ColorSpace`. Esta clase define una serie de constantes simbólicas que representan esos espacios de color. Algunas de esas constantes se muestran en la tabla 14.12.

Tabla 14.12 Espacios de Color

Constante	Descripción
`CS_GRAY`	El espacio de color de tonos de grises predefinido.
`CS_LINEAR_RGB`	El espacio de color RGB predefinido.
`TYPE_CMY`	Cualquiera de los espacios de color CMY.
`TYPE_CMYK`	Cualquiera de los espacios de color CMYK.

Para obtener una instancia de la clase **`ColorSpace`** podemos utilizar uno de sus métodos `getInstance()` cuya sintaxis se muestra en la tabla 14.13.

Tabla 14.13 Método `getInstance` de la clase `ColorSpace`

`public static ColorSpace` **`getInstance`**`(int colorspace)` Regresa un objeto del tipo `ColorSpace` representando uno de los espacios de color predefinidos.

La operación para convertir los colores de un espacio de color a otro está representada por la clase `ColorConvertOp`. Los métodos de la clase `ColorConvertOp` se muestran en la tabla 14.14.

Tabla 14.14 Métodos de la clase `LookUpOp`.

`public` **`ColorConvertOp`**`(ColorSpace cspace, RenderingHints hints)` Construye un objeto del tipo `ColorConvertOp` a partir de un objeto del tipo `ColorSpace` y una sugerencia de despliegue, la cual puede ser nula.
`public final BufferedImage` **`filter`**`(BufferedImage src, BufferedImage dst)` Convierte el espacio de color de de la imagen bufereada dada por el parámetro `src`. Si el parámetro `dst` es null, se creará una nueva imagen con el modelo de color de la imagen fuente.

Conversión de una Imagen de Color a Tonos de Grises

El siguiente código convierte una imagen de color a una de tonos de grises.

```
    /**
     * Este metodo convierte una imagen de color a tonos de grises.
     * @param frame Ventana sobre la que se despliega el mensaje de error
     */
    public void toGrises(JFrame frame) {
        // verifica que haya una imagen lista para ser procesada
        if(!preparaImagenProcesar(frame))
            return;

        // Crea una operacion de conversion de colores para cambiar
        // de colores a tonos de grises
        ColorConvertOp op = new ColorConvertOp(ColorSpace.
                                    getInstance(ColorSpace.CS_GRAY), null);

        // Filtra la imagen usando la operacion
        pbi = op.filter(pbi, null);

        // Hace que la referencia dbi apunte a obi
        dbi = pbi;
    }
```

La figura 14.24 muestra la imagen original y la imagen en tonos de grises.

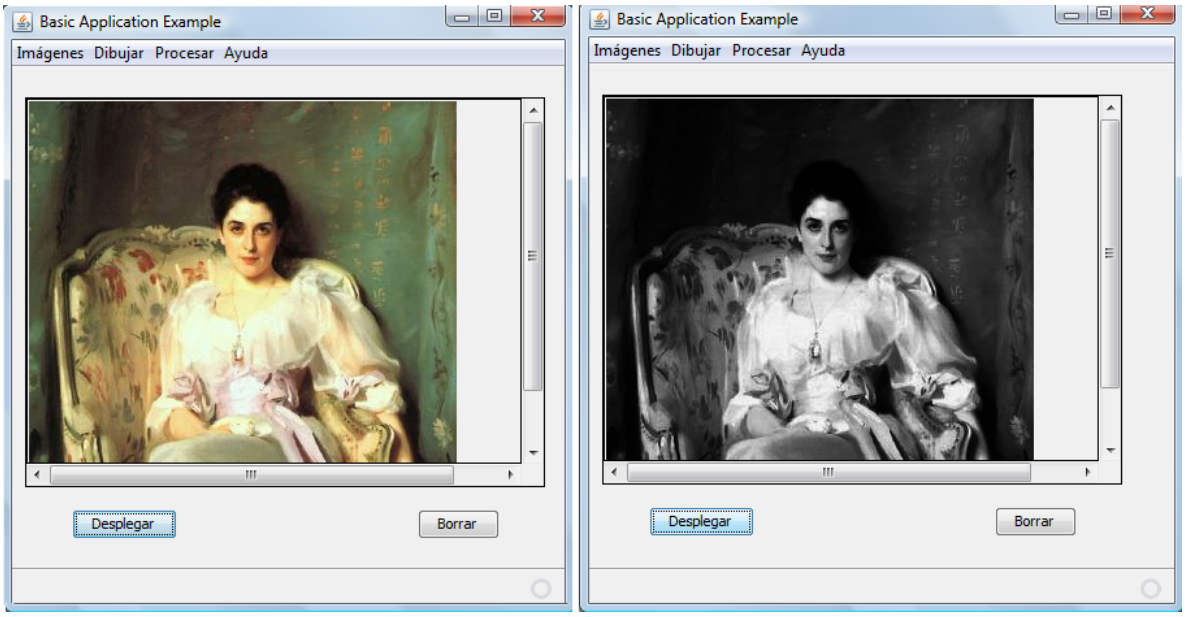

Figura 14.24. Izquierda: Imagen original, derecha: Imagen en tonos de grises.

Reescalamiento de Color

Esta operación se utiliza para ajustar la brillantez de una imagen multiplicando cada componente de color de cada pixel por un factor de escala.

La operación para reescalar la brillantez de una imagen está representada por la clase `RescaleOp`. La clase soporta un desplazamiento además del factor de

escala. Ese desplazamiento se le agrega al valor de la componente de color después de escalarla. Los métodos de la clase `ColorConvertOp` se muestran en la tabla 14.15.

Tabla 14.15 Métodos de la clase `RescaleOp`

`public` **`RescaleOp`**`(float scaleFactor, float offset, RenderingHints hints)` Construye un objeto del tipo `RescaleOp` con el factor de escala y desplazamiento deseados y una sugerencia de despliegue, la cual puede ser nula. El mismo escalamiento se aplica a todos las bandas de color.

`public` **`RescaleOp`**`(float[] scaleFactors, float[] offsets,`
`RenderingHints hints)`
 	
Construye un objeto del tipo RescaleOp con los factores de escala y desplazamientos deseados y una sugerencia de despliegue, la cual puede ser nula.

`public final BufferedImage` **`filter`**`(BufferedImage src, BufferedImage dst)` Escala la imagen bufereada dada por el parámetro `src`. Si el parámetro `dst` es null, se creará una nueva imagen con el modelo de color de la imagen fuente.

El siguiente código aumenta la brillantez de una imagen en un 20%.

```
    /**
     * Este metodo aumenta la brillantez de una imagen en un 20%,
     * Haciendo un reescalamiento de colores.
     * @param frame Ventana sobre la que se despliega el mensaje de error
     */
    public void aclara(JFrame frame) {
        // verifica que haya una imagen lista para ser procesada
        if(!preparaImagenProcesar(frame))
            return;

        // Crea una operacion de escalamiento para aclarar una imagen en
        // 20%
        RescaleOp op = new RescaleOp(1.2f, 0, null);

        // Filtra la imagen usando la operacion
        pbi = op.filter(pbi, null);

        // Hace que la referencia dbi apunte a obi
        dbi = pbi;
    }
```

La figura 14.25 muestra la imagen original y la imagen abrillantada un 20%

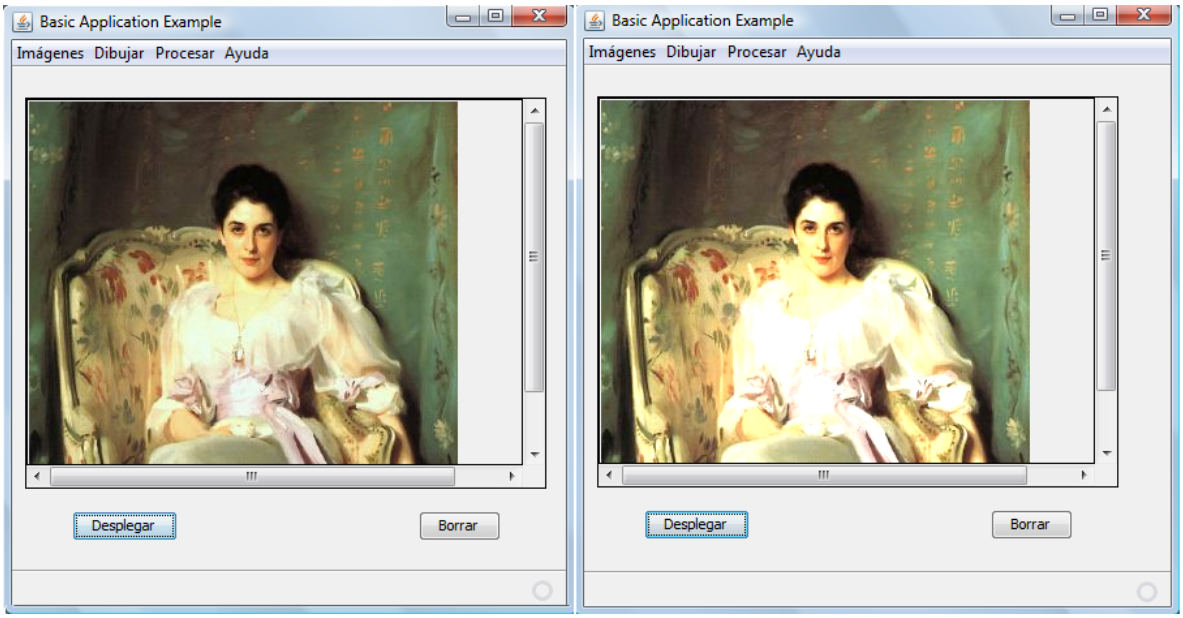

Figura 14.25. Izquierda: Imagen original, derecha: Imagen abrillantada 20%.

El siguiente código disminuye la brillantez de una imagen en un 20%.

```
    /**
     * Este metodo disminuye la brillantez de una imagen en un 20%,
     * Haciendo un reescalamiento de colores.
     * @param frame Ventana sobre la que se despliega el mensaje de error
     */
    public void oscurece(JFrame frame) {
        // verifica que haya una imagen lista para ser procesada
        if(!preparaImagenProcesar(frame))
            return;

        // Crea una operacion de escalamiento para oscurecer una imagen
        // en 20%
        RescaleOp op = new RescaleOp(0.8f, 0, null);

        // Filtra la imagen usando la operacion
        pbi = op.filter(pbi, null);

        // Hace que la referencia dbi apunte a obi
        dbi = pbi;
    }
```

La figura 14.26 muestra la imagen original y la imagen oscurecida un 20%

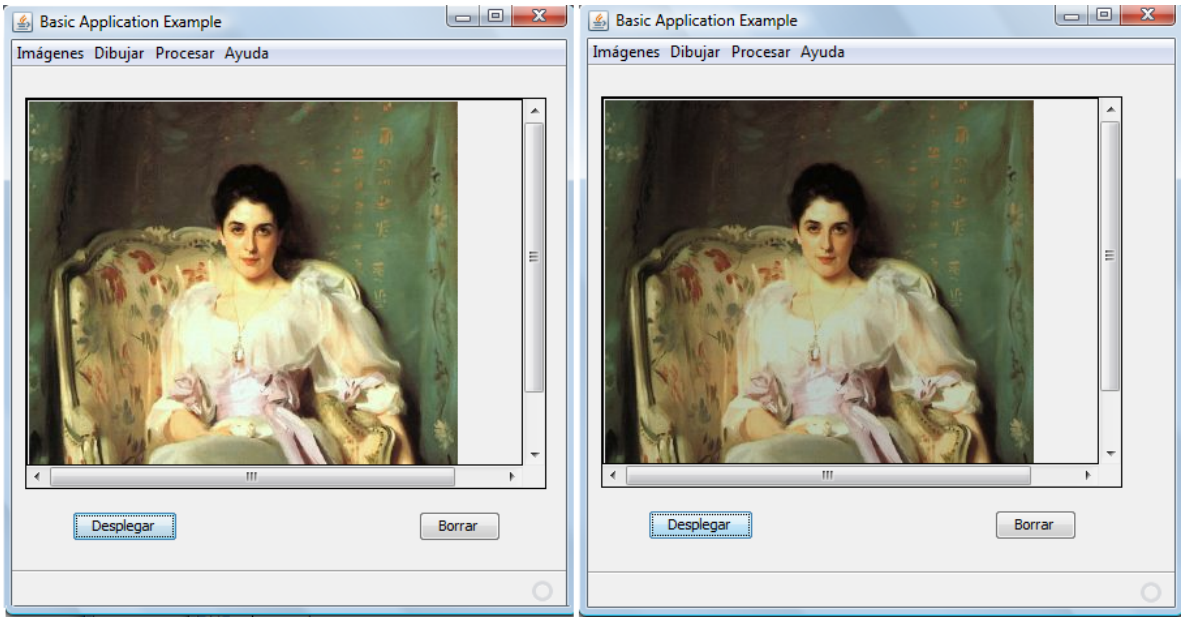

Figura 14.26. Izquierda: Imagen original, derecha: Imagen oscurecida 20%.

El siguiente código modifica la brillantez de una imagen, aclarando la banda roja un 20 %, oscureciendo la banda azul un 20% y dejando la banda verde sin modificar.

.

```
    /**
     * Este metodo modifica la brillantez de cada banda por separado
     * de una imagen, haciendo reescalamiento de colores.
     * @param frame Ventana sobre la que se despliega el mensaje de error
     */
    public void ajustaBrillantezBandas(JFrame frame) {
        // verifica que haya una imagen lista para ser procesada
        if(!preparaImagenProcesar(frame))
            return;

        // Crea una operacion de escalamiento de color para aclarar
        // la banda roja un 20 % y oscurecer la banda azul un 20%
        // dejando la banda verde sin modificar
        RescaleOp op = new RescaleOp(new float[]{1.2f, 1.0f, 0.8f},
                new float[]{32.0f, 0f, 0.0f}, null);

        // Filtra la imagen usando la operacion
        pbi = op.filter(pbi, null);

        // Hace que la referencia dbi apunte a obi
        dbi = pbi;
    }
```

La figura 14.27 muestra la imagen original y la imagen con las bandas modificadas en su brillantez por separado

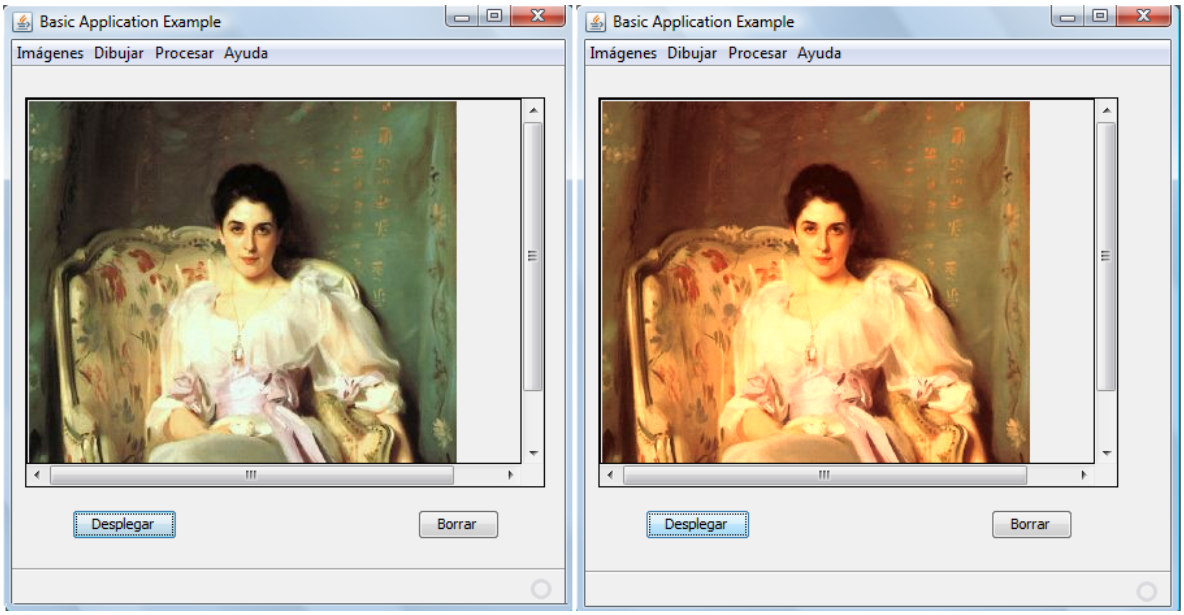

Figura 14.27. Izquierda: Imagen original, derecha: Imagen con las bandas modificadas en su brillantez por separado.

Transformación de Imágenes

Esta operación transforma una imagen escalandola, rotándola o deformándola.

Las transformaciones están representadas por la clase `AffineTransform`. Esta clase define una serie de métodos que regresan las diferentes transformaciones. Esos métodos se muestran en la tabla 14.16.

Tabla 14.16 Método de la clase `AffineTransform`

`public static AffineTransform` **`getRotateInstance`**`(double theta);` Regresa una transformación que rota la imagen alrededor del origen, el ángulo del parámetro en radianes.

`public static AffineTransform` **`getRotateInstance`**`(double theta,`
`double anchorx, double anchory);`
 	
Regresa una transformación que rota la imagen alrededor de las coordenadas 8x, y), el ángulo del parámetro en radianes.

`public static AffineTransform` **`getScaleInstance`**`(double sx, double sy)` Regresa una transformación que escala las dimensiones de la imagen en los valores de sus parámetros.

Tabla 14.16 Método de la clase `AffineTransform`. cont.

`public static AffineTransform` **`getShearInstance`**`(double shx, double shy)` Regresa una transformación que deforma las dimensiones de la imagen en los valores de sus parámetros.

La operación para transformar una imagen está representada por la clase `AffineTransformOp`. Los métodos de la clase `ColorConvertOp` se muestran en la tabla 14.17.

Tabla 14.17 Métodos de la clase `AffineTransformOp`.

`public` **`AffineTransformOp`**`(AffineTransform xform, int interpolationType)` Construye un objeto del tipo `AffineTransform`**`Op`** a partir de una transformación y un tipo de interpolación.
`public AffineTransform`**`Op`**`(AffineTransform xform, RenderingHints hints)` Construye un objeto del tipo `AffineTransform`**`Op`** a partir de una transformación y una sugerencia de despliegue, la cual puede ser nula.
`public final BufferedImage` **`filter`**`(BufferedImage src, BufferedImage dst)` Transforma la imagen bufereada dada por el parámetro `src`. Si el parámetro `dst` es null, se creará una nueva imagen con el modelo de color de la imagen fuente.

La tabla 14.18 muestra las constantes que definen los tipos de interpolació. Son constantes enteras estáticas.

Tabla 14.18 Tipos de interpolación

Constante	Descripción
`TYPE_NEAREST_NEIGHBOR`	Interpolación del tipo vecino más cercano..
`TYPE_BILINEAR`	Interpolación bilinear
`TYPE_BICUBIC`	Interpolación bicúbica.

Escalamiento de una Imagen

El siguiente código esacala una imagen aumentando su anchura en 20% y disminuyendo el alto en 20%

```
    /**
     * Este metodo aumenta el ancho de una imagen en 20% y disminuye
     * la altura en 20% haciendo una operacion de transformacion
     * @param frame Ventana sobre la que se despliega el mensaje de error
     */
    public void escala(JFrame frame) {
        // verifica que haya una imagen lista para ser procesada
```

```
        if(!preparaImagenProcesar(frame))
            return;

        // Crea una transformacion para aumentar el ancho de la imagen
        // en 20% y disminuir la altura en 20%
        AffineTransform at = AffineTransform.getScaleInstance(1.2, 0.8);
        RenderingHints rh = new RenderingHints(
                RenderingHints.KEY_INTERPOLATION,
                RenderingHints.VALUE_INTERPOLATION_BILINEAR);
        AffineTransformOp op = new AffineTransformOp(at, rh);

        // Filtra la imagen usando la operacion
        pbi = op.filter(pbi, null);

        // Hace que la referencia dbi apunte a obi
        dbi = pbi;
    }
```

La figura 14.28 muestra la imagen original y la imagen aumentada 20% en el eje de X y disminuida 20% en el eje Y.

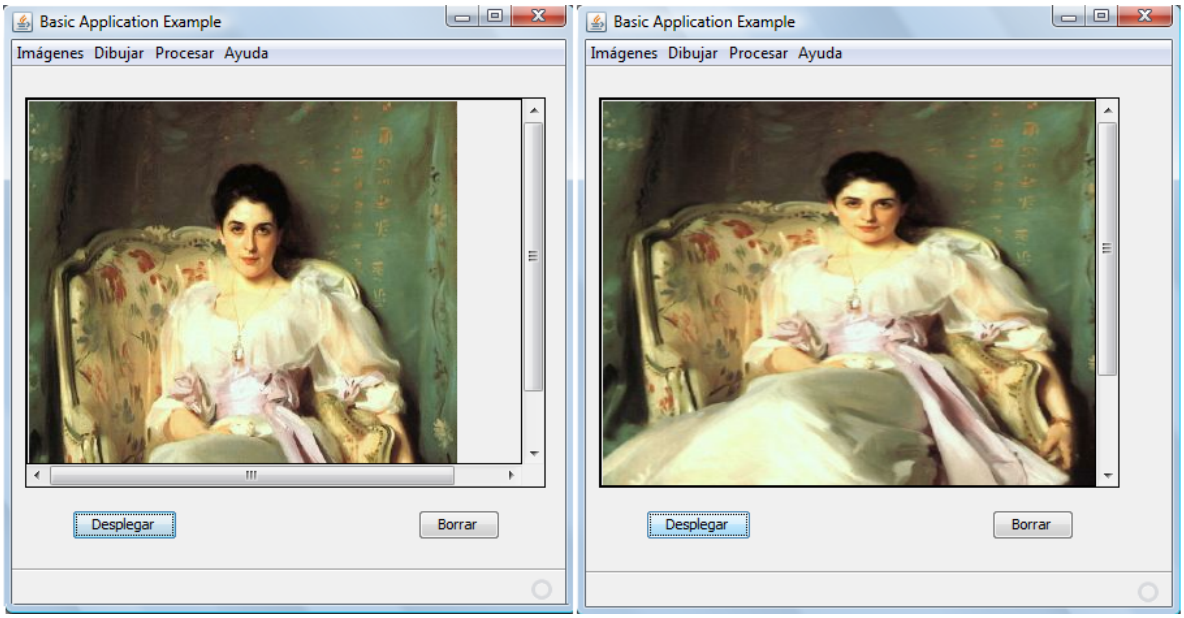

Figura 14.28. Izquierda: Imagen original, derecha: Imagen aumentada 20% en el eje de X y disminuida 20% en el eje Y.

El siguiente código rota una imagen 60° alrededor de su centro.

```
    /**
     * Este metodo rota una imagen en 60 grados alrededor de su centro.
     * haciendo una operacion de transformacion.
     * @param frame Ventana sobre la que se despliega el mensaje de error
```

```
     */
    public void rota(JFrame frame) {
        // verifica que haya una imagen lista para ser procesada
        if(!preparaImagenProcesar(frame))
            return;

        // Obtiene el tamaño de la imagen
        int altoImagen = pbi.getHeight();
        int anchoImagen = pbi.getWidth();

        // Crea una transformacion para rotar una imagen en 60 grados
        // alrededor de su centro
        AffineTransform at = AffineTransform.getRotateInstance(
                                Math.PI / 6, altoImagen/2, anchoImagen/2);
        RenderingHints rh = new RenderingHints(
                RenderingHints.KEY_INTERPOLATION,
                RenderingHints.VALUE_INTERPOLATION_BILINEAR);
        AffineTransformOp op = new AffineTransformOp(at, rh);

        // Filtra la imagen usando la operacion
        pbi = op.filter(pbi, null);

        // Hace que la referencia dbi apunte a obi
        dbi = pbi;
    }
```

La figura 14.29 muestra la imagen original rotada 60° alrededor de su centro.

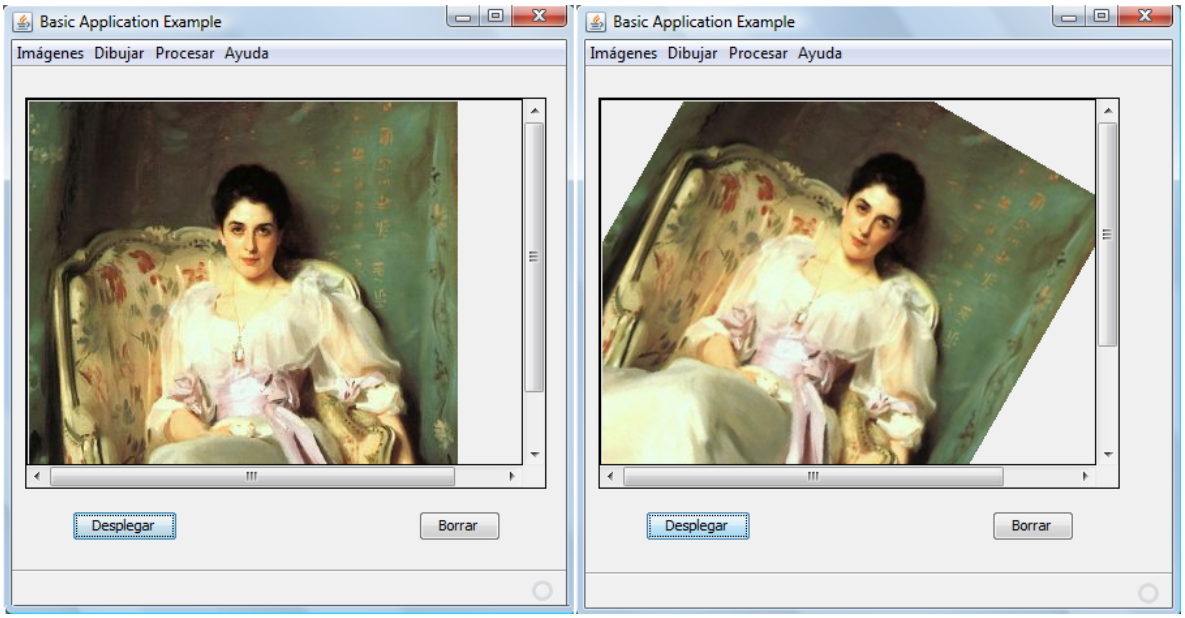

Figura 14.29. Izquierda: Imagen original, derecha: Imagen rotada 60° alrededor de su centro.